# Structurez vos données avec XML

**Par Ludovic ROLAND (Wapiti89)**




www.openclassrooms.com


*Dernière mise à jour le 7/10/2013*

# Sommaire



# Structurez vos données avec XML



Par Ludovic ROLAND (Wapiti89)

Mise à jour : 07/10/2013
Difficulté : Facile


## Vous souhaitez structurer les données manipulées ou échangées par vos programmes ou vos applications mobiles ?

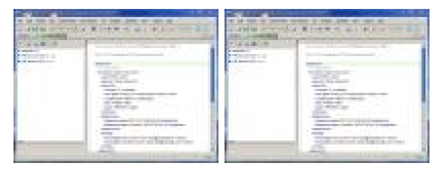
Ne cherchez plus ! XML va vous faciliter la vie !

XML est un langage de balisage générique qui permet de structurer des données afin qu'elles soient lisibles aussi bien par les humains que par des programmes de toute sorte. Il est souvent utilisé pour faire des échanges de données entre un programme et un serveur ou entre plusieurs programmes.

Pour vous donner un exemple concret, prenons un exemple d'actualité : celui d'une application téléphonique qui met à jour les données qu'elle contient. L'application demande à un serveur web les dernières informations dont il dispose. Après être allé les chercher, ce dernier doit les communiquer. C'est là qu'intervient le XML. Le serveur web se sert du XML pour structurer les informations qu'il doit renvoyer à l'application téléphonique. Lorsque cette dernière reçoit les informations ainsi structurées, elle sait comment les lire et les exploiter rapidement !

Dans ce cours destiné aux **débutants**, nous découvrirons ensemble comment écrire des documents XML. Puisque cette partie ne nous prendra pas énormément de temps, nous en profiterons pour découvrir ensemble tout ce qui tourne autour de l'univers du XML. Ainsi, nous verrons également comment :

- imposer une structure bien précise à nos documents XML ;
- les mettre en forme ;
- lire facilement les données contenues dans un document XML ;
- transformer les documents XML vers d'autres formats comme une page internet ou un fichier PDF ;



*Aperçu de fichiers que nous aurons l'occasion d'écrire dans ce tutoriel*

Vous l'aurez compris, le programme s'annonce chargé. C'est pourquoi je vous propose de commencer tout de suite ! 

# Partie 1 : Les bases du XML

Dans cette première partie, nous allons commencer par le tout début à savoir la structure d'un document XML.

Mais avant ça, nous allons nous attarder sur l'origine du XML, à quoi il sert et sur les bons outils à installer pour appréhender correctement ce tutoriel.

##  Qu'est-ce que le XML ?

Voici donc le tout premier chapitre de la longue série que va comporter ce tutoriel sur XML !

Tout au long de ce cours nous parlerons du XML et des technologies qui gravitent autours. Je vous propose pour ce premier chapitre de revenir un peu plus en détail sur cette technologie, son origine et son objectif.

### Qu'est ce que le XML ?

### Première définition

**Citation**

> Le XML ou e**X**tensible **M**arkup **L**anguage est un langage informatique de balisage générique.

Cette définition est à mes yeux un peu barbare et technique. C'est pourquoi je vous propose de décortiquer les mots-clefs.

#### *Un langage informatique*

Je suis sûr que vous n'êtes pas sans savoir qu'en informatique, il existe plusieurs centaines de langages destinés à des utilisations très diverses.

En vulgarisant un peu (beaucoup), il est possible de les regrouper dans 3 grosses catégories :

- les langages de programmation ;
- les langages de requête ;
- les langages de description.

Les **langages de programmation** permettent de créer des programmes, des applications mobiles, des sites internet, des systèmes d'exploitation, etc. Certains langages de programmation sont extrêmement populaires. En Mai 2012, les 10 langages de programmation les plus populaires étaient le C, le Java, le C++, l'Objective-C, le C#, le PHP, le Basic, le Python, le Perl et le Javascript.

Les **langages de requêtes** permettent quant à eux d'interroger des structures qui contiennent des données. Parmi les langages de requête les plus connus, on peut par exemple citer le SQL pour les bases de données relationnelles, le SPARQL pour les graphes RDF et les ontologies OWL ou encore le XQuery pour les documents XML.

Enfin, les **langages de description** permettent de décrire et structurer un ensemble de données selon un jeu de règles et des contraintes définies. On peut par exemple utiliser ce type de langage pour décrire l'ensemble des livres d'une bibliothèque, ou encore la liste des chansons d'un CD, etc. Parmi les langages de description les plus connus, on peut citer le SGML, le XML ou encore le HTML.

#### *Un langage de balisage générique*

Un langage de balisage est un langage qui s'écrit grâce à des balises. Ces balises permettent de structurer de manière hiérarchisée et organisée les données d'un document.

Si vous ne savez pas ce qu'est une balise, ne vous inquiétez pas, nous reviendrons sur ce terme un peu plus loin dans le cours.


Finalement, le terme **générique** signifie que nous allons pouvoir créer nos propres balises. Nous ne sommes pas obligés d'utiliser un ensemble de balises existantes comme c'est par exemple le cas en HTML.

## Une nouvelle définition

Suite aux explications du dessus, je vous propose que l'on écrive une nouvelle définition du langage XML bien moins technique que la première donnée dans ce cours.

Voici celle que je vous propose : **le langage XML est un langage qui permet de décrire des données à l'aide de balises et de règles que l'on peut personnaliser.**

Je me doute que certaines notions peuvent vous sembler abstraites, mais ne vous inquiétez pas, tout sera expliqué dans la suite de ce tutoriel.

## Origine et objectif du XML

## La petite histoire du XML

Vous vous doutez bien que le langage XML n'a pas été créé pour s'amuser. En effet, sa création avait pour objectif de répondre à un besoin très précis : **l'échange de données**.

Dans les débuts d'internet, les ordinateurs et les programmes échangeaient des données via des fichiers. Mais malheureusement, ces fichiers avaient bien souvent des règles de formatage qui leur étaient propres. Par exemple, les données étaient séparées des points, des virgules, des espaces, des tirets, etc.

Le problème avec ce système est qu'il fallait sans cesse adapter les programmes au format du fichier ce qui représentait une charge de travail importante.

Il a donc fallu régler ce problème assez vite. Le langage **SGML** ou **S**tandard **G**eneralized **M**arkup **L**anguage est alors né. C'est un langage puissant, extensible et standard qui permet de décrire à l'aide de balises un ensemble de données. Le problème est que ce langage très complexe n'est pas forcément compatible pour effectuer des échanges sur le web.

Un groupe d'informaticiens ayant de l'expérience dans le SGML et le web a alors décidé de se pencher sur le sujet. Le langage **XML** est né. Le XML 1.0 est devenu une recommandation du W3C (le "*World Wild Web Consortium*") le 10 février 1998.

Depuis, les spécifications du langage ont évolué, et la version 1.1 est publiée le 4 février 2004. C'est pourtant la version 1.0 du langage XML qui est la plus utilisée encore aujourd'hui et c'est cette version que nous allons étudier au cours de ce tutoriel.

## Les objectifs du XML

Comme nous l'avons vu, l'objectif du XML est de faciliter les échanges de données entre les machines. Mais à ça s'ajoute un autre objectif important : **décrire les données de manière aussi bien compréhensible par les hommes qui écrivent les documents XML que par les machines qui les exploitent.**

Le XML se veut également compatible avec le web afin que les échanges de données puissent se faire facilement à travers le réseau internet.

Finalement, le XML se veut standardisé, simple, mais surtout extensible et configurable afin que n'importe quel type de données puisse être décrit.
Pour résumer ce premier chapitre, peu technique mais très important, nous retiendrons que le XML est un langage de balisage générique qui permet de structurer des données dans l'objectif de les partager.



# Les bons outils

Maintenant que vous êtes un peu plus familier avec le XML et que son utilité est plus claire dans vos esprits, je vous propose de jeter un coup d'œil à certains outils qui pourront nous être utiles pour rédiger nos documents XML.

Les outils proposés dans ce tutoriel, nous permettront d'être plus productif. En effet, tous proposent des fonctions clefs pour gagner en productivité comme par exemple une fonction de coloration syntaxique, de validation, de vérification ou encore d'exploitation.

Je ne peux que vous encourager à les essayer et d'adopter celui qui vous correspond le plus !

## L'éditeur de texte

Il faut savoir que dans le fond, un document XML n'est en réalité qu'un simple document texte. C'est pourquoi, il est tout à fait possible d'utiliser un éditeur de texte pour la rédaction de nos documents XML !

### Sous Windows

Sous Windows, un éditeur de texte portant le nom de **Bloc-notes** est généralement installé par défaut. En théorie il est suffisant et fonctionnel. Dans la pratique, les fonctionnalités qu'il offre sont limitées et des options comme la coloration syntaxique ou la numérotation des lignes manquent.

Je vous propose donc d'utiliser **Notepad++** qui est parfait pour ce que nous voulons faire puisqu'il permet de pallier les manques du Bloc-notes. Il s'agit d'un logiciel gratuit, n'hésitez donc pas une seule seconde à le télécharger !

Télécharger Notepad++

Je ne vais pas détailler la procédure d'installation qui est classique pour un logiciel tournant sous Windows.

Une fois installé, lancez le logiciel. Vous devriez avoir une fenêtre semblable à la figure suivante.

Afin d'adapter la coloration syntaxique au langage XML, ils vous suffit de sélectionner `Langage` dans la barre de menu puis `XML` dans la liste.

Lorsque vous enregistrerez vos documents, il suffira alors de préciser comme extension "`.xml`" pour conserver la coloration syntaxique d'une fois sur l'autre.

## Sous GNU/Linux

Par défaut, les distributions Linux sous souvent livrées avec de très bons éditeurs de texte. Si vous aimez la console, vous pouvez par exemple utiliser **nano**, **emacs**, **vi** ou encore **vim**.
Si vous préférez les interfaces graphiques, je vous conseille d'utiliser l'excellent **gedit** qui normalement doit-être installé par défaut.
Si jamais ce n'est pas le cas, la commande suivante vous permettra de l'installer en quelques instants :

**Code : Console**

```
sudo apt-get install gedit
```

Une fois ouvert vous devriez avoir quelque chose comme ça (voir la figure suivante).

*Photo issue du site officiel du projet gedit*

Afin d'adapter la coloration syntaxique au langage XML, ils vous suffit de sélectionner `Affichage` dans la barre de menu puis `Mode de coloration` et finalement choisir le `XML` dans la liste.

Lorsque vous enregistrerez vos documents, il suffira alors de préciser comme extension "`.xml`" pour conserver la coloration syntaxique d'une fois sur l'autre.

### Sous MAC OS X

Pour les utilisateurs du système d'exploitation d'Apple, je vous conseille de vous tourner vers **jEdit**.
Vous pouvez le télécharger ici. La figure suivante vous donne un aperçu de son interface.

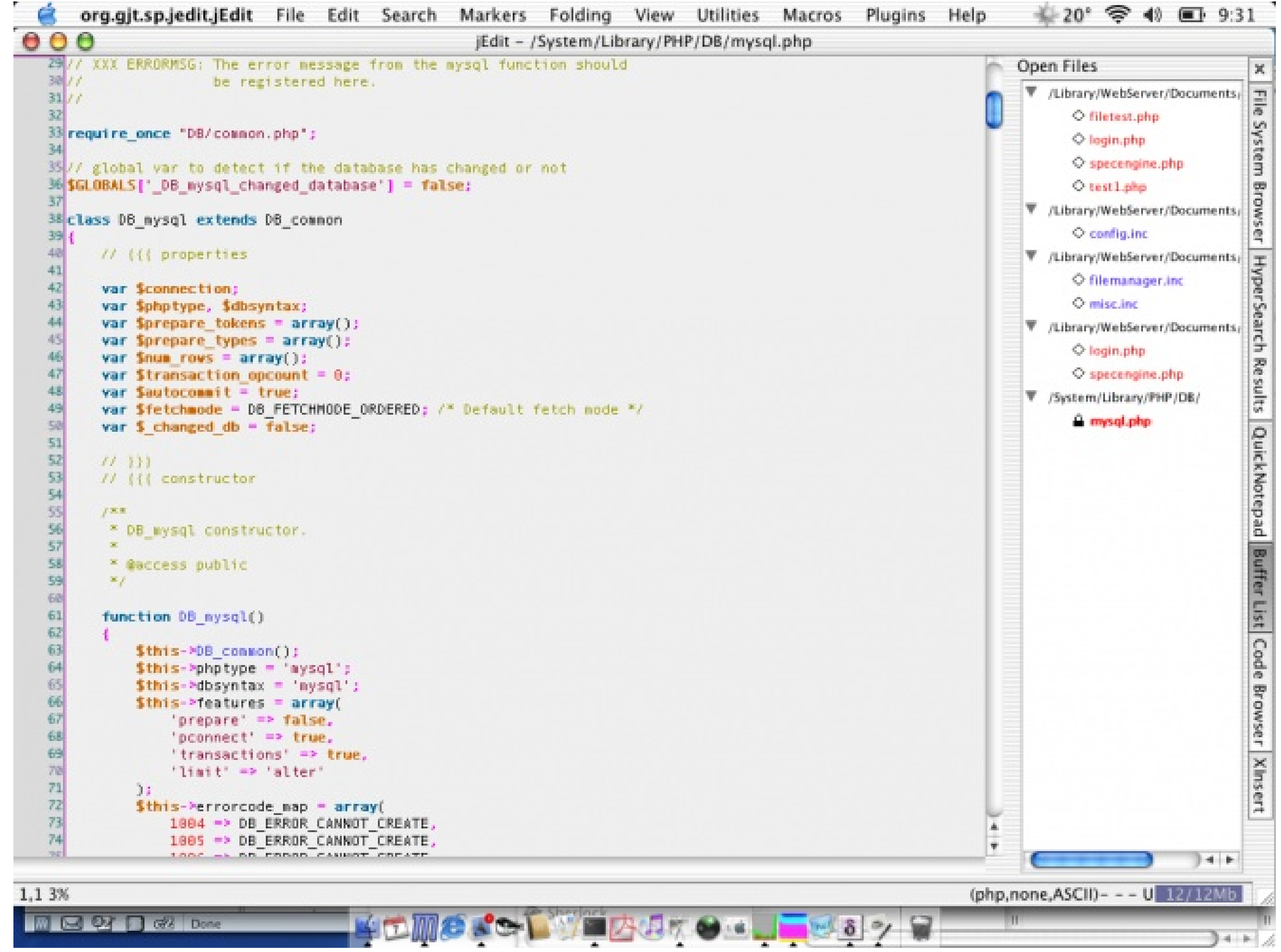



*Photo issue du site officiel du projet jEdit*

### EditiX

**Editix** est un éditeur XML qui fonctionne sur les plateformes Windows, GNU/Linux ou Mac OS X. En plus de la coloration syntaxique essentielle à l'écriture de documents XML, ce logiciel nous offre tout un tas d'outils qui nous seront utiles dans la suite de ce tutoriel comme par exemple la validation des documents.

#### La version payante

Il existe plusieurs versions de ce logiciel. La dernière en date est **EditiX 2012**.
Cette version complète est payante, mais plusieurs licences sont disponibles. Les étudiants peuvent par exemple bénéficier de 50% de réduction.

Télécharger EditiX 2012

#### La version gratuite

Heureusement pour les pauvres Zéros fauchés que nous sommes, une version gratuite existe ! Il s'agit d'une version allégée d'**Editix 2010**. Notez bien que cette version est réservée à un usage non-commercial !

Télécharger EditiX 2010, Lite Version

Puisque c'est cette version que je vais en partie utiliser dans la suite du tutoriel, je vous propose de détailler la procédure de mise en place du logiciel sous GNU/Linux. Je ne détaille pas la procédure d'installation sous Windows et MAC OS X puisqu'elle est des plus classique.

## La mise en place sous GNU/Linux

### *Téléchargement et installation*

Enregistrez l'archive sur votre bureau puis lancez votre plus beau terminal afin de débuter la procédure.
Commencez par déplacer l'archive `editix-free-2010.tar.gz` fraîchement téléchargée dans le répertoire `/opt/` via la commande :

**Code : Console**

```
sudo mv ~/Bureau/editix-free-2010.tar.gz /opt/
```

Déplacez vous maintenant dans le dossier `/opt/` via la commande :

**Code : Console**

```
cd /opt/
```

Nous allons maintenant extraire les fichiers de l'archive que nous avons téléchargée. Pour ce faire, vous pouvez utiliser la commande :

**Code : Console**

```
sudo tar xvzf editix-free-2010.tar.gz
```

Un dossier nommé `editix` doit alors être apparu. Il contient les fichiers que nous venons d'extraire.
Vous pouvez alors supprimer l'archive via la commande :

**Code : Console**

```
sudo rm editix-free-2010.tar.gz
```

On pourrait choisir de s'arrêter là et de lancer le logiciel en ligne de commande se rendant dans le répertoire `/opt/editix/bin/` et en exécutant le script `run.sh` via la commande :

**Code : Console**

```
./run.sh
```

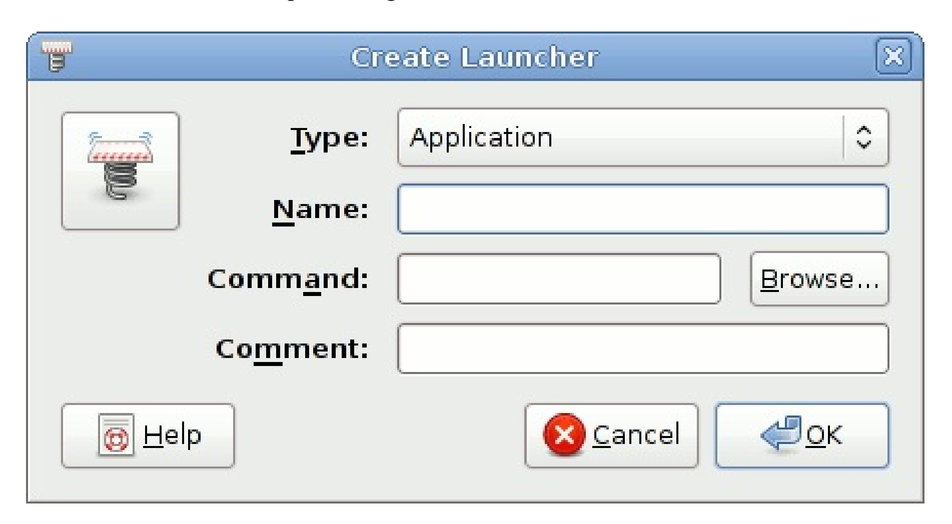
Mais pour plus de confort, je vous propose de créer un **launcher**.

### *Création du launcher*

Pour ce faire, faites un clic droit sur le menu `Applications` de votre tableau de bord (ou sur le `Menu` si vous êtes sur une LMDE par exemple) et cliquez sur `Editer le menu`.

Dans la colonne de gauche choisissez le menu `Programmation` puis cliquez sur le bouton `Nouvel élément` dans la colonne de droite.

Une fenêtre devrait alors s'afficher, comme indiqué à la figure suivante.




Remplissez le formulaire avec les informations suivantes :

- `Type` : Application
- `Nom` : Editix
- `Commande` : /opt/editix/bin/run.sh

Finalisez la création du launcher en cliquant sur le bouton `Valider`.
Editix devrait maintenant apparaître dans vos applications et plus particulièrement dans le menu programmation.

Quel que soit votre système d'exploitation, voici à la figure suivante ce à quoi doit ressembler la fenêtre du logiciel après son lancement.

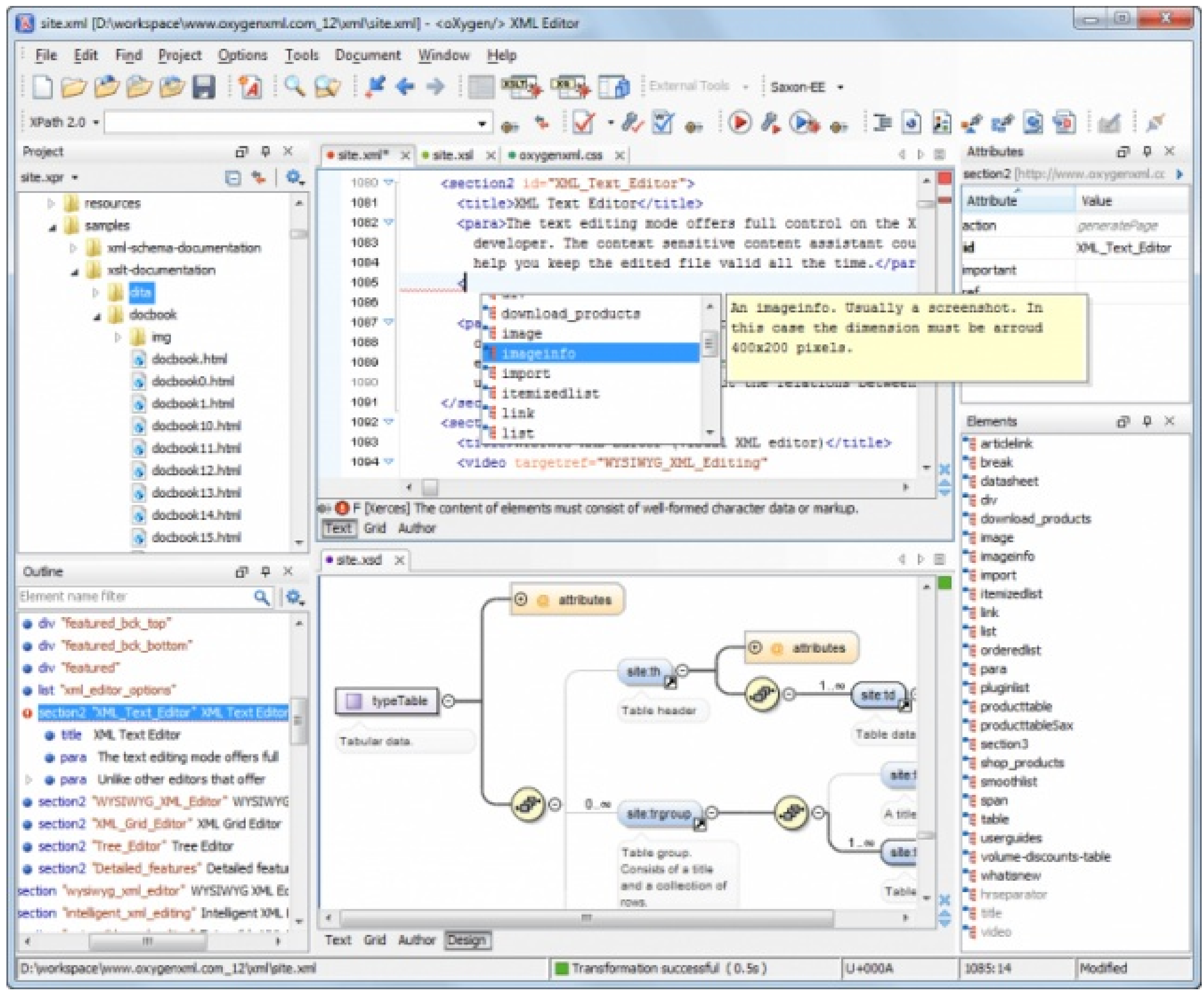
Image utilisateur

## <oXygen/> XML Editor

Je vais conclure cette présentation avec le logiciel **<oXygen/> XML Editor** qui comme Editix est multiplateformes.
Il n'existe pas de version gratuite du logiciel, mais il reste possible de le tester gratuitement pendant 30 jours. Comme pour **Editix** , **<oXygen/> XML Editor** propose plusieurs types de licences. Ainsi, les étudiants peuvent obtenir des réductions très intéressantes.

Télécharger <oXygen/> XML Editor

Voici à la figure suivante un exemple d'écran récupéré sur le site internet de l'éditeur du logiciel.




Maintenant que vous êtes équipé jusqu'aux dents, nous sommes prêts à commencer notre étude du langage XML !

#  Les éléments de base

C'est maintenant que la pratique commence ! Dans ce chapitre, nous allons découvrir ensemble les bases du XML.

Au programme donc de ce chapitre :

- les balises ;
- les attributs ;
- les commentaires.

## Les balises

Dans le tout premier chapitre, je vous définissais le langage XML comme un langage informatique de balisage. En effet, les balises sont les éléments de base d'un document XML. Une balise porte un nom qui est entouré de **chevrons**. Une balise commence donc par un < et se termine par un >. Par exemple : `<balise>` définit une balise qui s'appelle "**balise**".

En XML, on distingue 2 types de balises : les **balises par paires** et les **balises uniques**.

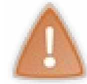
## Les balises par paires

### *Définition*

Les balises par paires sont composées en réalité de 2 balises que l'on appelle **ouvrantes** et **fermantes**.
La balise ouvrante commence par < et se termine par > tandis que la balise fermante commence par </ et se termine par >.
Par exemple :

**Code : XML**

```
<balise></balise>
```



Il est extrêmement important que les balises ouvrantes et fermantes aient **exactement** le même nom. XML est sensible à la casse (c'est-à-dire qu'il fait la distinction entre les majuscules et les minuscules) !
Toute balise ouverte doit impérativement être fermée. C'est une règle d'or !

Bien évidemment, on peut mettre "des choses" entre ces balises. On parle alors de **contenu**.
Par exemple :

**Code : XML**

```
<balise>Je suis le contenu de la balise</balise>
```

### *Quelques règles*

Une balise par paires ne peut pas contenir n'importe quoi : elle peut contenir une **valeur simple** comme par exemple une chaîne de caractères, un nombre entier, un nombre décimal, etc.

**Code : XML**

```
<balise1>Ceci est une chaîne de caractères</balise1>
<balise2>10</balise2>
<balise3>7.5</balise3>
```

Une balise en paires peut également contenir une **autre balise**. On parle alors d'**arborescence**.

**Code : XML**

```
<balise1>
  <balise2>10</balise2>
</balise1>
```



Faites cependant très attention, si une balise peut en contenir une autre, il est cependant interdit de les chevaucher. L'exemple suivant n'est pas du XML !

**Code : XML**

```
<balise1><balise2>Ceci est une chaîne de
caractères</balise1></balise2>
```

Finalement, une balise en paires peut contenir un mélange de valeurs simples et de balises comme en témoigne l'exemple suivant :

**Code : XML**

```
<balise1>
  Ceci est une chaîne de caractères
  <balise2>10</balise2>
  7.5
</balise1>
```

## Les balises uniques

Une balise unique est en réalité une balise en paires qui n'a pas de contenu.
Vous le savez, les informaticiens sont des fainéants. Ainsi, plutôt que de perdre du temps à ouvrir et fermer des balises sans rien écrire entre, une syntaxe un peu spéciale a été mise au point :

**Code : XML**

```
<balise />
```

## Les règles de nommage des balises

Ce qui rend le XML générique, c'est la possibilité de créer votre propre langage balisé. Ce langage balisé, comme son nom l'indique, est un langage composé de balises sauf qu'en XML, c'est vous qui choisissez leurs noms.

L'exemple le plus connu des langages balisés de type XML est très certainement le xHTML qui est utilisé dans la création de sites internet.

Il y a cependant quelques règles de nommage à respecter pour les balises de votre langage balisé :

- les noms peuvent contenir des lettres, des chiffres ou des caractères spéciaux ;
- les noms ne peuvent pas débuter par un nombre ou un caractère de ponctuation ;
- les noms ne peuvent pas commencer par les lettres XML (quelle que soit la casse) ;
- les noms ne peuvent pas contenir d'espaces ;

- on évitera les caractères **-**, **;** **.** < et > qui peuvent être mal interprétés dans vos programmes.

## Les attributs

### Définition

Il est possible d'ajouter à nos balises ce qu'on appelle des **attributs**. Tout comme pour les balises, c'est vous qui en choisissez le nom.
Un attribut peut se décrire comme une option ou une donnée cachée. Ce n'est pas l'information principale que souhaite transmettre la balise, mais il donne des renseignements supplémentaires sur son contenu.

Pour que ce soit un peu plus parlant, voici tout de suite un exemple :

**Code : XML**

```xml
<prix devise="euro">25.3</prix>
```

Dans l'exemple ci-dessus, l'information principale est le prix. L'attribut `devise` nous permet d'apporter des informations supplémentaires sur ce prix, mais ce n'est pas l'information principale que souhaite transmettre la balise `<prix/>`.

Une balise peut contenir 0 ou plusieurs attributs. Par exemple :

**Code : XML**

```xml
<prix devise="euro" moyen_paiement="chèque">25.3</prix>
```

### Quelques règles

Tout comme pour les balises, quelques règles sont à respecter pour les attributs :

- les règles de nommage sont les mêmes que pour les balises ;
- la valeur d'un attribut doit impérativement être délimitée par des guillemets, simples ou doubles ;
- dans une balise, un attribut ne peut-être présent qu'une seule fois.

## Les commentaires

Avant de passer à la création de notre premier document XML, j'aimerais vous parler des **commentaires**.

Un commentaire est un texte qui permet de donner une indication sur ce que l'on fait. Ils vous permettent d'annoter votre fichier et d'expliquer une partie de celui-ci.

En XML, les commentaires ont une syntaxe particulière. C'est une balise unique qui commence par **<!--** et qui se termine par **-->**.

**Code : XML**

```xml
<!-- Ceci est un commentaire ! -->
```

Voyons tout de suite sur un exemple concret :

**Code : XML**

```xml
<!-- Description du prix -->
<prix devise="euro">12.5</prix>
```

C'est sûr que sur cet exemple les commentaires semblent un peu inutiles... Mais je vous assure qu'ils vous seront d'une grande

aide pendant la rédaction de longs documents XML ! 

Pour résumer ce chapitre, nous venons de voir que 2 types de balises existent : les balises par paires, qui s'ouvrent et se ferment et les balises uniques que l'on peut considérer comme une balise par paires sans contenu.

Au cours de ce chapitre, nous avons également vu que les balises peuvent contenir des attributs et que de manière générale, un document XML peut contenir des commentaires.

#  Votre premier document XML

Jusqu'à maintenant, nous avons découvert les éléments de base du XML, mais vous ignorez encore comment écrire un document XML. Ne vous inquiétez pas, ce chapitre a pour objectif de corriger ce manque !

Dans ce chapitre, je vous propose donc de nous attaquer à tout ce qui se rattache à l'écriture d'un document XML. Nous reviendrons sur la structure générale d'un document et de nouvelles notions clefs avant d'utiliser le logiciel EditiX pour écrire notre premier document XML en bonne et due forme !

## Structure d'un document XML

Un document XML peut être découpé en 2 parties : le **prologue** et le **corps**.

### Le prologue

Il s'agit de la première ligne de votre document XML. Il donne des informations de traitement.

Voici à quoi va ressembler notre prologue dans cette première partie du tutoriel :

**Code : XML**

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
```

Comme vous pouvez le remarquer, le prologue est une balise unique qui commence par **<?xml** et qui se termine par **?>**.
Si vous ne comprenez pas cette ligne, pas de panique ! Nous allons tout décortiquer ensemble.

#### *La version*

Dans le prologue, on commence généralement par indiquer la version de XML que l'on utilise pour décrire nos données. Pour rappel, il existe actuellement 2 versions : 1.0 et 1.1.

À noter que le prologue n'est obligatoire que depuis la version 1.1, mais il est plus que conseillé de le mettre quand même lorsque vous utilisez la version 1.0.

La différence entre les 2 versions est une amélioration dans le support des différentes versions de l'Unicode. Sauf si vous souhaitez utiliser des caractères chinois dans vos documents XML, il conviendra d'utiliser la version 1.0 qui est encore aujourd'hui la version la plus utilisée.

#### *Le jeu de caractères*

La seconde information de mon prologue est `encoding="UTF-8"`.

Il s'agit du jeu de caractères utilisé dans mon document XML. Par défaut, l'encodage de XML est l'UTF-8, mais si votre éditeur de texte enregistre vos documents en ISO8859-1, il suffit de la changer dans le prologue :

**Code : XML**

```
<?xml version = "1.0" encoding="ISO8859-1" standalone="yes" ?>
```

#### *Un document autonome*

La dernière information présente dans le prologue est `standalone="yes"`.

Cette information permet de savoir si votre document XML est autonome ou si un autre document lui est rattaché.

Il est encore un peu tôt pour vous en dire plus. Nous reviendrons sur cette notion dans la partie 2 du tutoriel. Pour le moment acceptez le fait que nos documents sont tous autonomes.

## Le corps

Le corps d'un document XML est constitué de l'ensemble des balises qui décrivent les données. Il y a cependant une règle très importante à respecter dans la constitution du corps : **une balise en paires unique doit contenir toutes les autres**. Cette balise est appelée **élément racine** du corps.

Voyons tout de suite un exemple :

**Code : XML**

```
<racine>
  <balise_paire>texte</balise_paire>
  <balise_paire2>texte</balise_paire2>
  <balise_paire>texte</balise_paire>
</racine>
```

Bien évidemment, lorsque vous créez vos documents XML, le but est d'être le plus explicite possible dans le nommage de vos balises. Ainsi, le plus souvent, la balise racine aura pour mission de décrire ce quelle contient.

Si je choisis de décrire un répertoire, je peux par exemple nommer mes balises comme dans l'exemple suivant :

**Code : XML**

```
<repertoire>
  <personne>Bernard</personne>
  <personne>Patrick</personne>
</repertoire>
```

## Un document complet

Un document XML certes simple mais complet pourrait donc être le suivant :

**Code : XML**

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>
<repertoire>
  <personne>Bernard</personne>
  <personne>Patrick</personne>
</repertoire>
```

## Un document bien formé

Quand vous entendrez parler de XML, vous entendrez souvent parler de **document XML bien formé** ou **well-formed** en anglais.

Cette notion décrit en fait un document XML conforme aux règles syntaxiques décrites tout au long de cette première partie du tutoriel.

On peut résumer un document XML bien formé à un document XML avec une syntaxe correcte, c'est-à-dire :

- s'il s'agit d'un document utilisant la version 1.1 du XML, le prologue est bien renseigné ;
- le document XML ne possède qu'une seule balise racine ;
- le nom des balises et des attributs est conforme aux règles de nommage ;
- toutes les balises en paires sont correctement fermées ;
- toutes les valeurs des attributs sont entre guillemets simples ou doubles ;
- les balises de votre document XML ne se chevauchent pas, il existe une arborescence dans votre document.

Si votre document XML est bien formé, félicitation, il est exploitable ! Dans le cas contraire, votre document est inutilisable.

## Utilisation d'EditiX

Alors c'est bien beau, mais depuis le début on parle du XML, mais finalement nous n'avons toujours pas utilisé les logiciels du chapitre 2.

Dans cette partie, nous allons créer notre premier document XML grâce au logiciel EditiX et vérifier qu'il est bien formé.
Pour cet exemple, je vous propose d'utiliser le document complet que nous avons construit auparavant :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<repertoire>
  <personne>Robert</personne>
  <personne>John</personne>
</repertoire>
```

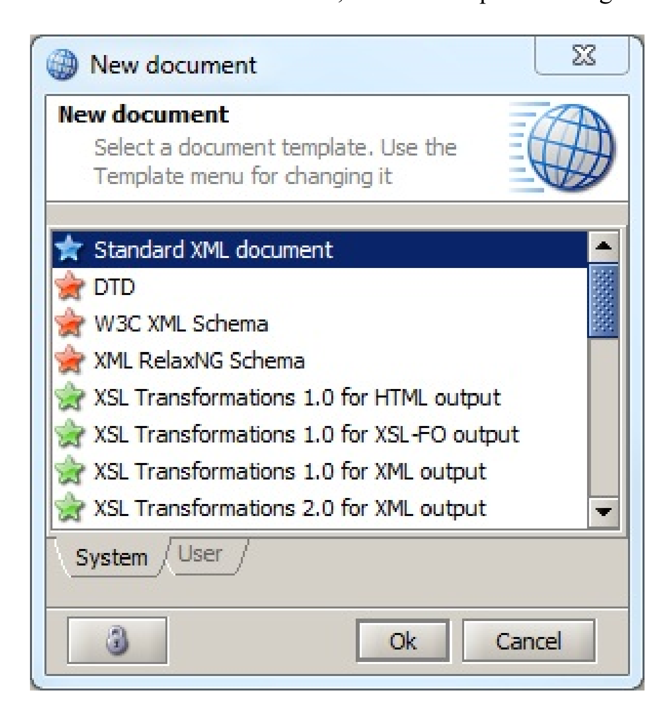
### Créer un nouveau document

Commencez par lancer EditiX.

Pour créer un nouveau document, vous pouvez cliquer sur l'icône suivante :



puis sélectionner dans la barre de menu **File** puis **New** ou encore utiliser le raccourci clavier Ctrl + N.
Dans la liste qui s'affiche, sélectionnez **Standard XML document**, comme indiqué sur la figure suivante.




Surprise ! Votre document XML n'est pas vierge. Voici ce que vous devriez avoir :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- New document created with EditiX at Fri May 18 00:11:02 CEST
2012 -->
```

Comme vous pouvez le constater, EditiX s'est chargé pour vous d'écrire le prologue de notre document XML. Il s'est également chargé d'écrire un petit commentaire pour votre rappeler la date et l'heure de création de votre document.

Puisque notre document sera autonome, vous pouvez modifier le prologue pour l'indiquer :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

## Vérification du document

Nous pouvons vérifier dès maintenant si notre document est bien formé. Pour se faire, vous pouvez cliquer sur l'icône suivante :



et sélectionner dans la barre de menu **XML** puis **Check this document** ou encore utiliser le raccourci clavier Ctrl + K.
Vous devriez alors avoir une erreur. La ligne où se situe l'erreur est représentée par un rectangle aux bords rouges sur notre espace de travail, comme indiqué sur la figure suivante.

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<!-- New document created with EditiX at Fri May 18 00:11:02 CEST 2012 -->

Nous avons donc une erreur à la ligne 6 de notre document. Pour en savoir plus sur notre erreur, il suffit de regarder en bas de l'écran, voici sur la figure suivante ce que vous devriez avoir.

In **1 Message(s)** / Click on each error for highlighting **( ctrl-shift-up / ctrl-shift-down )**

A well-formed document requires a root element.

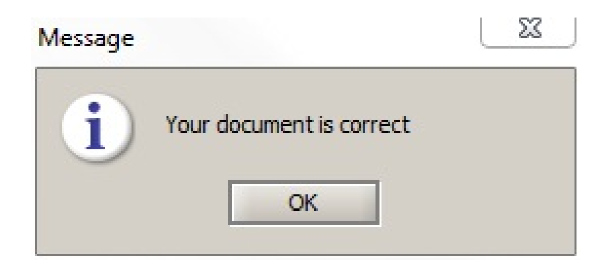
Pour ceux qui ne parlent pas anglais, voici ce que dit le message : « *Un document bien formé nécessite un élément racine.* »

Il manque donc un élément racine. Complétons tout de suite notre document avec les éléments suivant :

**Code : XML**

```
<repertoire>
  <personne>Robert</personne>
  <personne>John</personne>
</repertoire>
```

Lancer la vérification de votre document. Vous devriez avoir le message suivant à l'écran (voir la figure suivante).




Félicitation, votre document est bien formé !

## L'indentation

Il est possible de demander à Editix d'indenter automatiquement vos documents une fois écrits.
Pour ce faire, sélectionnez dans la barre de menu **XML** puis **Format** et **Pretty format (default)** ou utilisez le raccourci clavier Ctrl + R.
Dans ce même menu, vous pouvez accéder aux paramètres concernant la tabulation.

## L'arborescence du document

Editix met à votre disposition un outil fort sympathique qui vous permet de visualiser l'arborescence du document en cours d'édition (voir la figure suivante).




On sait ainsi que notre répertoire contient 2 personnes. La première s'appelle Robert et la seconde John.

## Enregistrer votre document

Il est maintenant temps de clore ce chapitre en enregistrant votre document XML.

Pour ce faire, vous pouvez cliquer sur l'icône suivante :

<image>http://uploads.siteduzero.com/files/379001_380000/
379797.png</image>

ou bien sélectionner dans la barre de menu `Save` puis `Save` ou encore utiliser le raccourci clavier Ctrl + S.

Dans la fenêtre de dialogue qui vient de s'ouvrir, choisissez l'emplacement ou vous souhaitez stocker votre fichier XML, tapez son nom et cliquez sur `Enregistrer`.
Croyez moi ou pas, mais si vous lisez ce message, c'est que vous maitrisez les bases du langage XML !

La suite de ce tutoriel sera l'occasion de voir plusieurs concepts rattachés au langage XML, mais à cette étape du tutoriel, vous êtes déjà capable de faire beaucoup ! 😉

Avant de passer à la seconde partie, je vous propose de tester les connaissances acquises au cours de cette première partie grâce à petit TP.

#  TP : structuration d'un répertoire

Voici donc le premier TP de ce tutoriel ! L'objectif de ces chapitres un peu particulier est de vous faire pratiquer. L'objectif est de vous permettre de restituer toute la théorie que vous avez lu au cours des chapitres précédents. C'est selon moi, indispensable pour s'assurer que vous avez bien compris ce qui est expliqué.

Dans ce premier TP, l'objectif est de vous montrer une utilisation concrète de structuration de données via XML.

## L'énoncé

Le but de ce TP est de créer un document XML structurant les données d'un répertoire.

Votre répertoire doit comprendre au moins 2 personnes. Pour chaque personne on souhaite connaître les informations suivantes :

- son sexe (homme ou femme) ;
- son nom ;
- son prénom ;
- son adresse ;
- un ou plusieurs numéros de téléphone (téléphone portable, fixe, bureau, etc.) ;
- une ou plusieurs adresses e-mail (adresse personnelle, professionnelle, etc.).

Je ne vous donne aucune indication concernant le choix des balises, des attributs et de l'arborescence à choisir pour une raison très simple : lorsqu'on débute en XML, le choix des attributs, des balises et de l'arborescence est assez difficile.

Le but est vraiment de vous laisser chercher et vous pousser à vous poser les bonnes questions sur l'utilité d'une balise, d'un attribut, etc.

## Une solution

Je vous fais part de ma solution ; notez bien que ce n'est qu'une solution parmi les multiples solutions possibles !

**Code : XML**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <adresse>
            <numero>7</numero>
            <voie type="impasse">impasse du chemin</voie>
            <codePostal>75015</codePostal>
            <ville>PARIS</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
        <emails>
            <email type="personnel">john.doe@wanadoo.fr</email>
            <email type="professionnel">john.doe@societe.com</email>
        </emails>
    </personne>

    <!-- Marie POPPINS -->
    <personne sexe="feminin">
        <nom>POPPINS</nom>
        <prenom>Marie</prenom>
        <adresse>
            <numero>28</numero>
            <voie type="avenue">avenue de la république</voie>
            <codePostal>13005</codePostal>
            <ville>MARSEILLE</ville>
            <pays>FRANCE</pays>
        </adresse>
```

```
            <telephones>
                <telephone type="professionnel">04 05 06 07
08</telephone>
            </telephones>
            <emails>
                <email type="professionnel">contact@poppins.fr</email>
            </emails>
        </personne>
    </repertoire>
```

## Quelques explications

### Le sexe

Comme vous pouvez le constater, j'ai fait le choix de renseigner le sexe dans un attribut de la balise `<personne/>` et non d'en faire une balise à part entière.

En effet, cette information est (je pense) plus utile à l'ordinateur qui lira le document qu'à un humain. En effet, contrairement à une machine, nous avons la capacité de déduire que John est un prénom masculin et Marie un prénom féminin. Cette information n'est donc pas cruciale pour les humains qui lisent le fichier.

### L'adresse

Il est important que vos documents XML aient une arborescence logique. C'est pourquoi j'ai décidé de représenter l'adresse postale par une balise `<adresse />` qui contient les informations détaillées de l'adresse de la personne comme le numéro dans la rue, la voie, le pays, etc.

J'ai également fait le choix d'ajouter un attribut **type** dans la balise `<voie />`. Une nouvelle fois, cet attribut est destiné à être utilisé par une machine.

En effet, une machine qui traitera ce fichier, pourra facilement accéder au type de la voie sans avoir à récupérer le contenu de la balise `<voie/>` et tenter d'analyser s'il s'agit d'une impasse, d'une rue, d'une avenue, etc. C'est donc un gain de temps dans le cas d'un traitement des données.

### Numéros de téléphone et adresses e-mails

Encore une fois, dans un soucis d'arborescence logique, j'ai décidé de créer les blocs `<telephones />` et `<emails />` qui contiennent respectivement l'ensemble des numéros de téléphone et des adresses e-mail.

Pour chacune des balises `<telephone/>` et `<email/>`, j'ai décidé d'y mettre un attribut **type**. Cet attribut permet de renseigner si l'adresse e-mail ou le numéro de téléphone est par exemple professionnel ou personnel.

Bien qu'indispensable aussi bien aux humains qu'aux machines, cette information est placée dans un attribut car ce n'est pas l'information principale que l'on souhaite transmettre. Ici l'information principale reste le numéro de téléphone ou l'adresse e-mail et non son type.

# Partie 2 : Créez des définitions pour vos documents XML

Maintenant que vous connaissez le XML, je vous propose de passer à la suite ! Dans cette seconde partie, nous verrons ensemble comment définir des structures précises pour nos documents. Ça sera l'occasion de découvrir 2 nouvelles technologies : les DTD et les Schémas XML.



# Introduction aux définitions et aux DTD

Après avoir découvert le XML dans une première partie, nous n'allons pas nous arrêter là ! En effet, pour être tout à fait honnête avec vous, il très rare que les documents XML soient utilisés seuls. Ils sont généralement accompagnés d'un second fichier qui permet de définir une structure stricte : c'est ce qu'on appelle un fichier de définition.

Dans ce premier chapitre, je vous propose d'étudier en détail les définitions et de voir en quoi c'est une notion importante. Finalement, nous étudierons également l'une des deux technologies permettant d'écrire des définitions : les DTD.

## Qu'est-ce que la définition d'un document XML ?

Avant de foncer tête baissée dans la seconde partie de ce cours, il est indispensable de revenir sur quelques termes qui seront importants pour la suite de ce tutoriel !

### Quelques définitions

#### *Définition d'une définition*

Une définition d'un document XML est un ensemble de règles que l'on impose au document. Ces règles permettent de décrire la façon dont le document XML doit-être construit. Elles peuvent-être de natures différentes. Par exemple, ces règles peuvent imposer la présence d'un attribut ou d'une balise, imposer l'ordre d'apparition des balises dans le document ou encore imposer le type d'une donnée (nombre entier, chaîne de caractères, etc.).

#### *Un document valide*

Dans la partie précédente, nous avons vu ce qu'était un document **bien formé**. Cette seconde partie est l'occasion d'aller un peu plus loin et de voir le concept de document **valide**.

**Un document valide est un document bien formé conforme à une définition.** Cela signifie que le document XML respecte toutes les règles qui lui sont imposées dans les fameuses définitions.

### Pourquoi écrire des définitions ?

Vous vous demandez certainement à quoi servent ces définitions et pourquoi on les utilise, n'est-ce pas ?

Associer une définition à un document oblige une certaine rigueur dans l'écriture de vos données XML. C'est d'autant plus important lorsque plusieurs personnes travaillent sur un même document. La définition impose ainsi une écriture uniforme que toutes les personnes doivent respecter. On évite ainsi l'écriture d'un document anarchique et difficilement exploitable.



Exploitable oui ! Mais par qui ?

Le plus souvent par un programme informatique ! Vous pouvez par exemple écrire un programme informatique qui traite les données contenues dans un document XML respectant une définition donnée. Imposer une définition aux documents que votre programme exploite permet d'assurer un automatisme et un gain précieux de temps :

- le document n'est pas valide : je ne tente pas de l'exploiter ;
- le document est valide : je sais comment l'exploiter.

Pour terminer cette longue introduction, sachez que vous avez le choix entre deux technologies pour écrire les définitions de vos documents XML : les DTD ou les schémas XML.

# Définition d'une DTD

## Une définition rapide

Une **Document Type Definition** ou en français une **Définition de Type de Document**, souvent abrégé **DTD**, est la première technologie que nous allons étudier pour écrire les définitions de nos documents XML.

Comme déjà précisé dans la longue introduction de cette seconde partie, le but est d'écrire une définition à nos documents XML, c'est-à-dire construire un ensemble de règles qui vont régir la construction du document XML.

Grâce à l'ensemble de ces règles, on va ainsi définir l'architecture de notre document XML et la hiérarchie qui existe entre les balises de celui-ci. Ainsi, on pourra préciser l'enchaînement et le contenu des balises et des attributs contenus dans le document XML.

Finalement, sachez qu'avec les DTD vous ne pourrez pas toujours tout faire, la technologie commençant à vieillir. Mais vu qu'elle est encore beaucoup utilisée, il est indispensable qu'elle soit étudiée dans ce tutoriel !

## Où écrire les DTD ?

Tout comme les fichiers XML, les DTD s'écrivent dans des fichiers.

Nous reviendrons dessus un peu plus tard, mais sachez dès à présent qu'il existe 2 types de DTD : les **DTD externes** et les **DTD internes**.

Les règles des **DTD internes** s'écrivent directement dans le fichier XML qu'elles définissent tandis que les règles des **DTD externes** sont écrites dans un fichier séparé portant l'extension **.dtd** .

Maintenant que vous en savez un peu plus, je vous propose de rentrer dans le vif du sujet.

# Les éléments

## La syntaxe

Pour définir les règles portant sur les **balises**, on utilise le mot clef `ELEMENT`.

**Code : XML**

```
<!ELEMENT balise (contenu)>
```

Une règle peut donc se découper en 3 mots clefs : `ELEMENT`, `balise` et `contenu`.

## Retour sur la balise

Le mot-clef `balise` est à remplacer par le nom de la balise à laquelle vous souhaitez appliquer la règle.
Pour exemple, reprenons une balise du TP de la partie 1 :

**Code : XML**

```
<nom>DOE</nom>
```

On écrira alors :

**Code : XML**

```
<!ELEMENT nom (contenu)>
```

## Retour sur le contenu

Cet emplacement a pour vocation de décrire ce que doit contenir la balise : est-ce une autre balise ou est-ce une valeur ?

### *Cas d'une balise en contenant une autre*

Par exemple, regardons la règle suivante :

**Code : XML**

```
<!ELEMENT personne (nom)>
<!-- suite de la DTD -->
```

Cette règle signifie que la balise `<personne />` contient la balise `<nom />`.

Le document XML respectant cette règle ressemble donc à cela :

**Code : XML**

```
<personne>
    <nom>John DOE</nom>
</personne>
```



Nous n'avons défini aucune règle pour la balise `<nom/>`. Le document n'est en conséquence pas valide. **En effet, dans une DTD, il est impératif de décrire tout le document sans exception.** Des balises qui n'apparaissent pas das la DTD ne peuvent pas être utilisées dans le document XML.

### *Cas d'une balise contenant une valeur*

Dans le cas où notre balise contient une **valeur simple**, on utilisera la mot clef `#PCDATA`
Une valeur simple désigne par exemple une chaîne de caractères, un entier, un nombre décimal, un caractère, etc.
En se basant sur l'exemple exemple précédent :

**Code : XML**

```
<personne>
    <nom>John DOE</nom>
</personne>
```

Nous avions déjà défini une règle pour la balise `<personne/>` :

**Code : XML**

```
<!ELEMENT personne (nom)>
```

Nous pouvons maintenant compléter notre DTD en ajoutant une règle pour la balise **<nom/>**. Par exemple, si l'on souhaite que cette balise contienne une valeur simple, on écrira :

**Code : XML**

```
<!ELEMENT nom (#PCDATA)>
```

Au final, la DTD de notre document XML est donc la suivante :

**Code : XML**

```
<!ELEMENT personne (nom)>
<!ELEMENT nom (#PCDATA)>
```

### *Cas d'une balise vide*

Il est également possible d'indiquer qu'une balise ne contient rien grâce au mot-clef `EMPTY`.
Prenons les règles suivantes :

**Code : XML**

```
<!ELEMENT personne (nom)>
<!ELEMENT nom EMPTY>
```

Le document XML répondant à la définition DTD précédente est le suivant :

**Code : XML**

```
<personne>
    <nom />
</personne>
```



À noter que lors de l'utilisation du mot clef `EMPTY`, l'usage des parenthèses n'est pas obligatoire !

### *Cas d'une balise contenant n'importe quoi*

Il nous reste un cas à voir : celui d'une balise qui peut contenir n'importe quoi. C'est à dire que notre balise peut contenir une autre balise, une valeur simple ou tout simplement être vide. Dans ce cas, on utilise le mot-clef `ANY`.

Prenons la règle suivante :

**Code : XML**

```
<!ELEMENT personne (nom)>
<!ELEMENT nom ANY>
```

Les documents XML suivants sont bien valides :

**Code : XML**

```
<!-- valeur simple -->
<personne>
    <nom>John DOE</nom>
</personne>

<!-- vide -->
<personne>
    <nom />
</personne>
```



Bien que le mot-clef `ANY` existe, il est souvent déconseillé de l'utiliser afin de restreindre le plus possible la liberté de rédaction du document XML.



Comme pour le mot-clef `EMPTY`, l'usage des parenthèses n'est pas obligatoire pour le mot-clef `ANY` !

## Structurer le contenu des balises

Nous allons voir maintenant des syntaxes permettant d'apporter un peu de généricité aux définitions DTD.
Par exemple, un répertoire contient généralement un nombre variable de personnes, il faut donc permettre au document XML d'être valide quel que soit le nombre de personnes qu'il contient.

### La séquence

Une séquence permet de décrire l'enchaînement imposé des balises. Il suffit d'indiquer le nom des balises en les séparant par des **virgules**.

**Code : XML**

```
<!ELEMENT balise (balise2, balise3, balise4, balise5, etc.)>
```

Prenons l'exemple suivant :

**Code : XML**

```
<!ELEMENT personne (nom, prenom, age)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT age (#PCDATA)>
```

Cette définition impose que la balise **`<personne />`** contienne obligatoirement les balises **`<nom />`**, **`<prenom />`** et

<age /> dans cet ordre.
Regardons alors la validité des documents XML qui suivent :

**Code : XML**

```
<!-- valide -->
<personne>
    <nom>DOE</nom>
    <prenom>John</prenom>
    <age>24</age>
</personne>

<!-- invalide -->
<!-- les balises ne sont pas dans le bon ordre -->
<personne>
    <prenom>John</prenom>
    <nom>DOE</nom>
    <age>24</age>
</personne>

<!-- invalide -->
<!-- il manque une balise -->
<personne>
    <prenom>John</prenom>
    <age>24</age>
</personne>

<!-- invalide -->
<!-- il y a une balise en trop qui plus est n'est pas déclarée -->
<personne>
    <nom>DOE</nom>
    <prenom>John</prenom>
    <age>24</age>
    <date>12/12/2012</date>
</personne>
```

## La liste de choix

Une liste de choix permet de dire qu'une balise contient l'une des balises décrites. Il suffit d'indiquer le nom des balises en les séparant par une **barre verticale**.

**Code : XML**

```
<!ELEMENT balise (balise2 | balise3 | balise4 | balise5 | etc.)>
```

Prenons l'exemple suivant :

**Code : XML**

```
<!ELEMENT personne (nom | prenom)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
```

Cette définition impose que la balise <personne /> contienne obligatoirement la balise <nom /> ou la balise <prenom />.

Regardons alors la validité des documents XML ci-dessous :

**Code : XML**

```
<!-- valide -->
<personne>
    <nom>DOE</nom>
</personne>

<!-- valide -->
<personne>
    <prenom>John</prenom>
</personne>

<!-- invalide -->
<!-- les 2 balises prenom et nom ne peuvent pas être présentes en
même temps. -->
<personne>
    <prenom>John</prenom>
    <nom>DOE</nom>
</personne>

<!-- invalide -->
<!-- il manque une balise -->
<personne />
```

## La balise optionnelle

Une balise peut-être optionnelle. Pour indiquer qu'une balise est optionnelle, on fait suivre son nom par un **point d'interrogation**.

**Code : XML**

```
<!ELEMENT balise (balise2, balise3?, balise4)>
```

Prenons l'exemple suivant :

**Code : XML**

```
<!ELEMENT personne (nom, prenom?)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
```

Cette définition impose que la balise `<personne />` contienne obligatoirement la balise `<nom />` puis éventuellement `<prenom />`.
Regardons alors la validité de ces documents XML :

**Code : XML**

```
<!-- valide -->
<personne>
    <nom>DOE</nom>
</personne>

<!-- valide -->
<personne>
    <nom>DOE</nom>
```

```
        <prenom>John</prenom>
    </personne>

    <!-- invalide -->
    <!-- l'ordre des balises n'est pas respecté -->
    <personne>
        <prenom>John</prenom>
        <nom>DOE</nom>
    </personne>
```

## La balise répétée optionnelle

Une balise peut-être répétée plusieurs fois bien qu'optionnelle. Pour indiquer une telle balise on fait suivre son nom par une **étoile**.

**Code : XML**

```
<!ELEMENT balise (balise2, balise3*, balise4)>
```

Soit l'ensemble de règles suivant :

**Code : XML**

```
<!ELEMENT repertoire (personne*)>
<!ELEMENT personne (nom, prenom)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
```

Cette définition impose que la balise `<repertoire />` contienne entre 0 et une infinité de fois la balise `<personne />`. La balise `<personne />` quant à elle doit obligatoirement contenir les balises `<nom />` et `<prenom />` dans cet ordre.

Regardons alors la validité des documents XML :

**Code : XML**

```
<!-- valide -->
<repertoire>
    <personne>
        <nom>DOE</nom>
        <prenom>John</prenom>
    </personne>
    <personne>
        <nom>POPPINS</nom>
        <prenom>Marie</prenom>
    </personne>
</repertoire>

<!-- valide -->
<repertoire>
    <personne>
        <nom>DOE</nom>
        <prenom>John</prenom>
    </personne>
</repertoire>

<!-- valide -->
```

```
<repertoire />

<!-- invalide -->
<!-- il manque la balise prenom dans la seconde balise personne-->
<repertoire>
    <personne>
        <nom>DOE</nom>
        <prenom>John</prenom>
    </personne>
    <personne>
        <nom>POPPINS</nom>
    </personne>
</repertoire>
```

## La balise répétée

Une balise peut-être répétée plusieurs fois. Pour indiquer une telle balise on fait suivre son nom par un **plus**.

**Code : XML**

```
<!ELEMENT balise (balise2, balise3+, balise4)>
```

Prenons l'exemple suivant :

**Code : XML**

```
<!ELEMENT repertoire (personne+)>
<!ELEMENT personne (nom, prenom)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>
```

Cette définition impose que la balise `<repertoire />` contienne **au minimum une fois** la balise `<personne />`. La balise `<personne />` quant à elle doit obligatoirement contenir les balises `<nom />` et `<prenom />` dans cet ordre.

Regardons alors la validité des documents XML suivants :

**Code : XML**

```
<!-- valide -->
<repertoire>
    <personne>
        <nom>DOE</nom>
        <prenom>John</prenom>
    </personne>
    <personne>
        <nom>POPPINS</nom>
        <prenom>Marie</prenom>
    </personne>
</repertoire>

<!-- valide -->
<repertoire>
    <personne>
        <nom>DOE</nom>
        <prenom>John</prenom>
    </personne>
```

```
</repertoire>

<!-- invalide -->
<!-- la balise personne doit être présente au moins une fois-->
<repertoire />
```

Nous venons de voir dans ce chapitre de nombreuses nouveautés, à savoir les prémices d'une nouvelle technologie, une nouvelle syntaxe et plein de mots clefs à retenir !

Je suis conscient que ce n'est pas facile à digérer, c'est pourquoi je vous encourage à relire plusieurs fois ce chapitre avant de passer à la suite afin que vous soyez certains d'avoir tout compris !

#  DTD : les attributs et les entités

Dans le chapitre précédent, nous avons vu comment décrire les balises de nos documents XML. Mais rappelez vous, une balise peut contenir ce qu'on appelle des attributs. C'est ce que nous allons voir au cours de ce chapitre.

Nous reviendrons également sur une notion dont je n'ai pas encore parlé : les entités. Je laisse un peu de suspense autour de la définition de cette notion et son utilisation.

## Les attributs

Dans le chapitre précédent, nous avons vu la syntaxe pour définir des règles sur les balises de nos documents XML. Vous allez voir que le principe est le même pour définir des règles à nos attributs.

### La syntaxe

Pour indiquer que notre règle porte sur un **attribut**, on utilise le mot clef `ATTLIST`. On utilise alors la syntaxe suivante :

**Code : XML**

```
<!ATTLIST balise attribut type mode>
```

Une règle peut donc se découper en 5 mots clefs : `ATTLIST`, `balise`, `attribut`, `type` et `mode`.

### Retour sur la balise et l'attribut

Il n'est nécessaire de s'attarder trop longtemps sur le sujet, il suffit simplement d'écrire le nom de la balise et de l'attribut concerné par la règle.

Par exemple, reprenons une balise du TP de la partie 1 :

**Code : XML**

```
<personne sexe="masculin" />
```

On écrira alors :

**Code : XML**

```
<!ATTLIST personne sexe type mode>
```

### Retour sur le type

Cet emplacement a pour vocation de décrire le **type** de l'attribut. Est-ce une valeur bien précise ? du texte ? un identifiant ?

#### *Cas d'un attribut ayant pour type la liste des valeurs possibles*

Nous allons étudier ici le cas d'un attribut ayant pour type une liste de valeurs. Les différentes valeurs possibles pour l'attribut sont séparées par une **barre verticale** .

**Code : XML**

```
<!ATTLIST balise attribut (valeur 1 | valeur 2 | valeur 3 | etc.)
```

```
mode>
```

Reprenons une nouvelle fois la balise `<personne />`. Nous avons vu que cette balise possède un attribut **sexe**. Nous allons ici imposer la valeur que peut prendre cette attribut : soit **masculin**, soit **féminin**.

Voici ce à quoi la règle portant sur l'attribut dans notre DTD doit ressembler :

**Code : XML**

```
<!ATTLIST personne sexe (masculin|féminin) mode>
```

Quelques exemple de documents XML possibles :

**Code : XML**

```
<!-- valide -->
<personne sexe="masculin" />

<!-- valide -->
<personne sexe="féminin" />

<!-- invalide -->
<personne sexe="autre" />
```

### *Cas d'un attribut ayant pour type du texte non "parsé"*

Derrière le terme "**texte non "parsé"**" se cache en fait la possibilité de mettre ce que l'on veut comme valeur : un nombre, une lettre, une chaîne de caractères, etc. Il s'agit de données qui ne seront pas analysées par le "parseur" au moment de la validation et/ou l'exploitation de votre document XML.

Dans le cas où notre attribut contient du **texte non "parsé"**, on utilise la mot clef `CDATA`.

**Code : XML**

```
<!ATTLIST balise attribut CDATA mode>
```

Soit la règle suivante :

**Code : XML**

```
<!ATTLIST personne sexe CDATA mode>
```

Notre document XML répondant à cette règle peut ressembler à cela :

**Code : XML**

```
<!-- valide -->
<personne sexe="masculin" />

<!-- valide -->
<personne sexe="féminin" />
```

```
<!-- valide -->
<personne sexe="autre" />

<!-- valide -->
<personne sexe="12" />
```

### *Cas d'un attribut ayant pour type un identifiant unique*

Il est tout à fait possible de vouloir qu'une balise possède un attribut permettant de l'identifier de manière unique.

Prenons par exemple l'exemple d'une course à pied. Dans le classement de la course, il y aura un **unique** vainqueur, un **unique** second et un **unique** troisième.

Pour indiquer que la valeur de l'attribut est unique, on utilise le mot clef **ID** comme **ID**entifiant.

**Code : XML**

```
<!ATTLIST balise attribut ID mode>
```

Prenons par exemple la règle suivante :

**Code : XML**

```
<!ATTLIST personne position ID mode>
```

Voici quelques exemples de documents XML :

**Code : XML**

```
<!-- valide -->
<personne position="POS-1" />
<personne position="POS-2" />
<personne position="POS-3" />

<!-- invalide -->
<personne position="POS-1" />
<personne position="POS-1" />
<personne position="POS-2" />
```

### *Cas d'un attribut ayant pour type une référence à un identifiant unique*

Il est tout à fait possible que dans votre document, un de vos attributs fasse référence à un identifiant. Cela permet souvent de ne pas écrire 100 fois les mêmes informations.

Par exemple, votre document XML peut vous servir à représenter des liens de parenté entre des personnes. Grâce aux références, nous n'allons pas devoir imbriquer des balises XML dans tous les sens pour tenter de représenter le père d'une personne ou le fils d'une personne.

Pour faire référence à un identifiant unique, on utilise le mot clef `IDREF`.

Prenons par exemple la règle suivante :

**Code : XML**

```
<!ATTLIST father id ID mode >
<!ATTLIST child id ID mode
father IDREF mode
>
```

Cette règle signifie que la balise personne a 2 attributs : **id** qui est l'identifiant unique de la personne et **father** qui fait référence une autre personne.

Illustrons immédiatement avec un exemple XML :

**Code : XML**

```
<!-- valide -->
<father id="PER-1" />
<child id="PER-2" father="PER-1" />

<!-- invalide -->
<!-- l'identifiant PER-0 n'apparaît nulle part -->
<father id="PER-1" />
<child id="PER-2" father="PER-0" />
```

Dans cet exemple, la personne **PER-2** a pour père la personne **PER-1**. On matérialise bien le lien entre ces 2 personnes.

## Retour sur le mode

Cet emplacement permet de donner une information supplémentaire sur l'attribut comme par exemple une indication sur son obligation ou sa valeur.

### *Cas d'un attribut obligatoire*

Lorsqu'on souhaite qu'un attribut soit obligatoirement renseigné, on utilise le mot clef `#REQUIRED`.

Par exemple, si l'on souhaite que le sexe d'une personne soit renseigné, on utilisera la règle suivante :

**Code : XML**

```
<!ATTLIST personne sexe (masculin|féminin) #REQUIRED>
```

Voici alors quelques exemple de documents XML possibles :

**Code : XML**

```
<!-- valide -->
<personne sexe="masculin" />

<!-- valide -->
<personne sexe="féminin" />

<!-- invalide -->
<personne />
```

### Cas d'un attribut optionnel

Si au contraire on souhaite indiquer qu'un attribut n'est pas obligatoire, on utilise le mot clef `#IMPLIED`.

Si l'on reprend l'exemple précédent, on peut indiquer qu'il n'est pas obligatoire de renseigner le sexe d'une personne via la règle suivante :

**Code : XML**

```
<!ATTLIST personne sexe CDATA #IMPLIED>
```

Voici alors quelques exemple de documents XML possibles :

**Code : XML**

```
<!-- valide -->
<personne sexe="masculin" />

<!-- valide -->
<personne sexe="féminin" />

<!-- valide -->
<personne sexe="15" />

<!-- valide -->
<personne />
```

### Cas d'une valeur par défaut

Il est également possible d'indiquer une valeur par défaut pour un attribut. Il suffit tout simplement d'écrire cette valeur en dur dans la règle.

Par exemple, il est possible d'indiquer qu'une personne dont l'attribut **sexe** n'est pas renseigné est un homme par défaut grâce à la règle suivante :

**Code : XML**

```
<!ATTLIST personne sexe CDATA "masculin">
```

Voici alors quelques exemple de documents XML possibles :

**Code : XML**

```
<!-- valide -->
<personne sexe="masculin" />

<!-- valide -->
<personne sexe="féminin" />

<!-- valide -->
<!-- l'attribut sexe vaut "masculin" -->
<personne />
```

### Cas d'une constante

Finalement, il est possible de rendre obligatoire un attribut et de fixer sa valeur grâce au mot clef `#FIXED` suivi de la dite valeur.

Cette situation peut par exemple se rencontrer lorsqu'on souhaite travailler dans une devise bien précise et qu'on souhaite qu'elle apparaisse dans le document.

Par exemple, la règle suivante permet d'indiquer que la devise doit obligatoirement apparaître et a pour seule valeur possible l'`euro`.

**Code : XML**

```
<!ATTLIST objet devise CDATA #FIXED "Euro">
```

Voici alors quelques exemple de documents XML possibles :

**Code : XML**

```
<!-- valide -->
<objet devise="Euro" />

<!-- invalide -->
<objet devise="Dollar" />

<!-- invalide -->
<objet />
```

# Les entités

Une autre notion assez importante à voir lorsqu'on parle de DTD est la notion d'**entité**.

## Définition

Une entité peut-être considérée comme un alias permettant de réutiliser des informations au sein du document XML ou de la définition DTD.

Au cours de ce chapitre, nous reviendrons sur les 3 types d'entités : les entités générales, les entités paramètres et les entités externes.

## Les entités générales

### *Définition*

Les entités générales sont les entités les plus simples. Elles permettent d'associer un alias à une information afin de l'utiliser dans le document XML.

### *La syntaxe*

Voyons tout de suite la syntaxe d'une entité générale :

**Code : XML**

```
<!ENTITY nom "valeur">
```

Pour utiliser une entité générale dans notre document XML, il suffit d'utiliser la syntaxe suivante :

**Code : XML**

```
&nom;
```

Afin d'illustrer un peu plus clairement mes propos, voyons tout de suite un exemple :

**Code : XML**

```
<!ENTITY samsung "Samsung">
<!ENTITY apple "Apple">

<telephone>
    <marque>&samsung;</marque>
    <modele>Galaxy S3</modele>
</telephone>
<telephone>
    <marque>&apple;</marque>
    <modele>iPhone 4</modele>
</telephone>
```

Au moment de son interprétation, les références aux entités seront remplacées par leurs valeurs respectives, ce qui donne une fois interprété :

**Code : XML**

```
<telephone>
    <marque>Samsung</marque>
    <modele>Galaxy S3</modele>
</telephone>
<telephone>
    <marque>Apple</marque>
    <modele>iPhone 4</modele>
</telephone>
```

## Les entités paramètres

### *Définition*

Contrairement aux entités générales qui apparaissent dans les documents XML, les entités paramètres n'apparaissent que dans les définitions DTD. Elles permettent d'associer un alias à une partie de la déclaration de la DTD.

### *La syntaxe*

Voyons tout de suite la syntaxe d'une entité paramètre :

**Code : XML**

```
<!ENTITY % nom "valeur">
```

Pour utiliser une entité paramètre dans notre DTD, il suffit d'utiliser la syntaxe suivante :

**Code : XML**

```
%nom;
```

Prenons par exemple cet exemple où des téléphones ont pour attribut une marque :

**Code : XML**

```
<telephone marque="Samsung" />
<telephone marque="Apple" />
```

Normalement, pour indiquer que l'attribut `marque` de la balise `<telephone/>` est obligatoire et qu'il doit contenir la valeur Samnsung ou Apple, nous devons écrire la règle suivante :

**Code : XML**

```
<!ATTLIST telephone marque (Samsung|Apple) #REQUIRED>
```

À l'aide d'une entité paramètre, cette même règle s'écrit de la façon suivante :

**Code : XML**

```
<!ENTITY % listeMarques "marque (Samsung|Apple) #REQUIRED">
<!ATTLIST telephone %listeMarques; >
```

Encore une fois, au moment de son interprétation, les références aux entités seront remplacées par leurs valeurs respectives.

## Les entités externes

### *Définition*

Il existe en réalité 2 types d'entités externes : les analysées et les non analysées. Dans le cadre de ce cours, nous nous limiterons aux entités externes analysées.

Les entités externes analysées ont sensiblement le même rôle que les entités générales, c'est à dire qu'elles permettent d'associer un alias à une information afin de l'utiliser dans le document XML. Mais dans le cas des entités externes analysées, les informations sont stockées dans un fichier séparés.

### *La syntaxe*

Voyons tout de suite la syntaxe d'une entité externe :

**Code : XML**

```
<!ENTITY nom SYSTEM "URI">
```

Pour utiliser une entité externe dans notre XML, il suffit d'utiliser la syntaxe suivante :

**Code : XML**

```
&nom;
```

Si l'on reprend notre premier exemple, voici ce que cela donne :

**Code : XML**

```
<!ENTITY samsung SYSTEM "samsung.xml">
<!ENTITY apple SYSTEM "apple.xml">

<telephone>
    &samsung;
    <modele>Galaxy S3</modele>
</telephone>
<telephone>
    &apple;
    <modele>iPhone 4</modele>
</telephone>
```

Le contenu des fichiers `samsung.xml` et `apple.xml` sera par exemple le suivant :

**Code : XML**

```
<!-- Contenu du fichier samsung.xml -->
<marque>Samsung</marque>

<!-- Contenu du fichier apple.xml -->
<marque>Apple</marque>
```

Au moment de son interprétation, les références aux entités seront remplacées par leurs valeurs respectives, ce qui donne une fois interprété :

**Code : XML**

```
<telephone>
    <marque>Samsung</marque>
    <modele>Galaxy S3</modele>
</telephone>
<telephone>
    <marque>Apple</marque>
    <modele>iPhone 4</modele>
</telephone>
```

Dans ce second chapitre, nous venons de voir comment définir les attributs de vos balises XML. Nous avons également découvert une nouvelle notion : les entités. Elles nous permettent de jouer les fainéants en réutilisant des éléments qui reviennent souvent dans nos documents.

#  DTD : où les écrire ?

Nous venons donc d'étudier au cours des derniers chapitres tout ce qu'il faut savoir ou presque sur les DTD. En effet, vous ignorez encore une information importante sur les DTD vous permettant de passer de la théorie à la pratique : où les écrire.

Ca sera également l'occasion de vous révéler qu'il existe en réalité plusieurs sortes de DTD.

## Les DTD internes

Comme je vous l'ai déjà précisé dans le premier chapitre de cette seconde partie, on distingue 2 types de DTD : les internes et les externes.

Commençons par étudier les internes !

### Définition

Une DTD interne est une DTD qui est écrite dans le même fichier que le document XML. Elle est généralement spécifique au document XML dans lequel elle est écrite.

### La Syntaxe

Une DTD interne s'écrit dans ce qu'on appelle le **DOCTYPE**. On le place sous le prologue du document et au dessus du contenu XML.

Voyons plus précisément la syntaxe :

**Code : XML**

```
<!DOCTYPE racine [ ]>
```

La DTD interne est ensuite écrite entre les **[]**. Dans ce **DOCTYPE**, le mot **racine** doit être remplacé par le nom de la balise qui forme la racine du document XML.

### Illustrons avec un exemple

Afin que tout cela vous paraisse moins abstrait, je vous propose de voir un petit exemple.

Prenons par exemple l'énoncé suivant :

**Citation : Enoncé**

> Une boutique possède plusieurs téléphones. Chaque téléphone est d'une certaine marque et d'un certain modèle représenté par une chaîne de caractère.

Un document XML répondant à cet énoncé est par exemple le suivant :

**Code : XML**

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>

<boutique>
  <telephone>
    <marque>Samsung</marque>
    <modele>Galaxy S3</modele>
  </telephone>

  <telephone>
    <marque>Apple</marque>
```

```
      <modele>iPhone 4</modele>
    </telephone>

    <telephone>
      <marque>Nokia</marque>
      <modele>Lumia 800</modele>
    </telephone>
  </boutique>
```

La définition DTD est la suivante :

**Code : XML**

```
<!ELEMENT boutique (telephone*)>
<!ELEMENT telephone (marque, modele)>
<!ELEMENT marque (#PCDATA)>
<!ELEMENT modele (#PCDATA)>
```

Le document XML complet avec la DTD interne sera par conséquent le suivant :

**Code : XML**

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>

<!DOCTYPE boutique [
<!ELEMENT boutique (telephone*)>
  <!ELEMENT telephone (marque, modele)>
  <!ELEMENT marque (#PCDATA)>
  <!ELEMENT modele (#PCDATA)>
]>

<boutique>
  <telephone>
    <marque>Samsung</marque>
    <modele>Galaxy S3</modele>
  </telephone>

  <telephone>
    <marque>Apple</marque>
    <modele>iPhone 4</modele>
  </telephone>

  <telephone>
    <marque>Nokia</marque>
    <modele>Lumia 800</modele>
  </telephone>
</boutique>
```

Maintenant que vous savez ce qu'est une DTD interne, passons à la DTD externe !

## Les DTD externes

### Définition

Une DTD externe est une DTD qui est écrite dans un autre document que le document XML. Si elle est écrite dans un autre document, c'est que souvent, elle est commune à plusieurs documents XML qui l'exploitent.

De manière générale, afin de bien séparer le contenu XML de sa définition DTD, on prendra l'habitude de séparer dans plusieurs

fichiers.

Un fichier contenant uniquement une DTD porte l'extension **.dtd**.

## La Syntaxe

L'étude de la syntaxe d'une DTD externe est l'occasion de vous révéler qu'il existe en réalité 2 types : les DTD externes **PUBLIC** et les DTD externes **SYSTEM**.

Dans les 2 cas et comme pour une DTD interne, c'est dans le **DOCTYPE** que cela se passe.

### *Les DTD externes PUBLIC*

Les DTD externes PUBLIC sont généralement utilisées lorsque la DTD est une norme. C'est par exemple cas dans les documents xHTML 1.0.

La syntaxe est la suivante :

**Code : XML**

```
<!DOCTYPE racine PUBLIC "identifiant" "url">
```

Si on l'applique à un document xHTML, on obtient alors le **DOCTYPE** suivant :

**Code : XML**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Pour être honnête, nous n'allons jamais utiliser les DTD externes PUBLIC dans ce tutoriel, c'est pourquoi je vous propose de passer immédiatement aux DTD externes SYSTEM.

### *Les DTD externes SYSTEM*

Une DTD externe SYSTEM permet d'indiquer au document XML l'adresse du document DTD. Cette adresse peut-être relative ou absolue.

Voyons plus précisément la syntaxe :

**Code : XML**

```
<!DOCTYPE racine SYSTEM "URI">
```

Afin d'illustrer mes propos, je vous propose de reprendre l'exemple de la boutique de téléphone que j'ai utilisé dans la partie sur la DTD interne.

Voici un rappel de l'énoncé :

**Citation : Enoncé**

> Une boutique possède plusieurs téléphones. Chaque téléphone est d'une certaine marque et d'un certain modèle, tous les 2 représentés par une chaîne de caractère.

Pour rappel, voici le fichier XML :

**Code : XML**

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>

<boutique>
  <telephone>
    <marque>Samsung</marque>
    <modele>Galaxy S3</modele>
  </telephone>

  <telephone>
    <marque>Apple</marque>
    <modele>iPhone 4</modele>
  </telephone>

  <telephone>
    <marque>Nokia</marque>
    <modele>Lumia 800</modele>
  </telephone>
</boutique>
```

Si la DTD ne change pas, elle doit cependant être placée dans un fichier à part, par exemple le fichier **doc1.dtd**. Voici son contenu :

**Code : XML**

```
<!ELEMENT boutique (telephone*)>
<!ELEMENT telephone (marque, modele)>
<!ELEMENT marque (#PCDATA)>
<!ELEMENT modele (#PCDATA)>
```

Le document XML complet avec la DTD externe sera alors le suivant (on part ici du principe que le fichier XML et DTD sont stockés au même endroit) :

**Code : XML**

```
<?xml version = "1.0" encoding="UTF-8" standalone="yes" ?>

<!DOCTYPE boutique SYSTEM "doc1.dtd">

<boutique>
  <telephone>
    <marque>Samsung</marque>
    <modele>Galaxy S3</modele>
  </telephone>

  <telephone>
    <marque>Apple</marque>
    <modele>iPhone 4</modele>
  </telephone>

  <telephone>
    <marque>Nokia</marque>
    <modele>Lumia 800</modele>
  </telephone>
</boutique>
```

## Retour sur le prologue

Dans la partie précédente de ce tutoriel, voici ce que je vous avais dit à propos du fait que nos documents XML soient autonomes ou non :

**Citation : Wapiti89**

> La dernière information présente dans le prologue est `standalone="yes"`.
>
> Cette information permet de savoir si votre document XML est autonome ou si un autre document lui est rattaché.
>
> Il est encore un peu tôt pour vous en dire plus. Nous reviendrons sur cette notion dans la partie 2 du tutoriel. Pour le moment acceptez le fait que nos documents sont tous autonomes.

Il est maintenant temps de lever le mystère !

Dans le cas d'une DTD externe, nos documents XML ne sont plus autonomes, en effet, ils font références à un autre fichier qui fournit la DTD. Afin que le document contenant la DTD soit bien pris en compte, nous devons l'indiquer en passant simplement la valeur de l'attribut `standalone` à "**no**".

Voici ce que cela donne :

**Code : XML**

```
<?xml version = "1.0" encoding="UTF-8" standalone="no" ?>

<!DOCTYPE boutique SYSTEM "doc1.dtd">

<boutique>
  <telephone>
    <marque>Samsung</marque>
    <modele>Galaxy S3</modele>
  </telephone>

  <telephone>
    <marque>Apple</marque>
    <modele>iPhone 4</modele>
  </telephone>

  <telephone>
    <marque>Nokia</marque>
    <modele>Lumia 800</modele>
  </telephone>
</boutique>
```

## Un exemple avec EditiX

Afin de terminer ce chapitre, je vous propose de voir ensemble comment écrire une DTD externe SYSTEM avec EditiX.
Pour faire simple, je vous propose se garder l'exemple précédent de la boutique de téléphone.

## Création du document XML

La création du document XML n'a rien de bien compliqué puisque nous l'avons déjà vu ensemble dans la partie précédente.
Pour ceux qui ne s'en souviennent pas, vous pouvez y jeter un coup d'œil ici.

Voici le document que vous devez écrire :

**Code : XML**

```
<?xml version = "1.0" encoding="UTF-8" standalone="no" ?>

<!DOCTYPE boutique SYSTEM "boutique.dtd">

<boutique>
  <telephone>
    <marque>Samsung</marque>
    <modele>Galaxy S3</modele>
  </telephone>

  <telephone>
    <marque>Apple</marque>
    <modele>iPhone 4</modele>
  </telephone>

  <telephone>
    <marque>Nokia</marque>
    <modele>Lumia 800</modele>
  </telephone>
</boutique>
```

Si vous essayez de lancer la vérification du document, vous devriez normalement avoir un message d'erreur, comme celui indiqué en figure suivante.

| In | **1 Message(s)** / Click on each error for highlighting **( ctrl-shift-up / ctrl-shift-down )** |
|---|---|
| | Can't load the DTD J:\ludovic\Documents\Programmation\XML\Tutoriel\boutique.dtd |

Ce message est pour le moment complètement normal puisque nous n'avons pas encore créé notre document DTD.

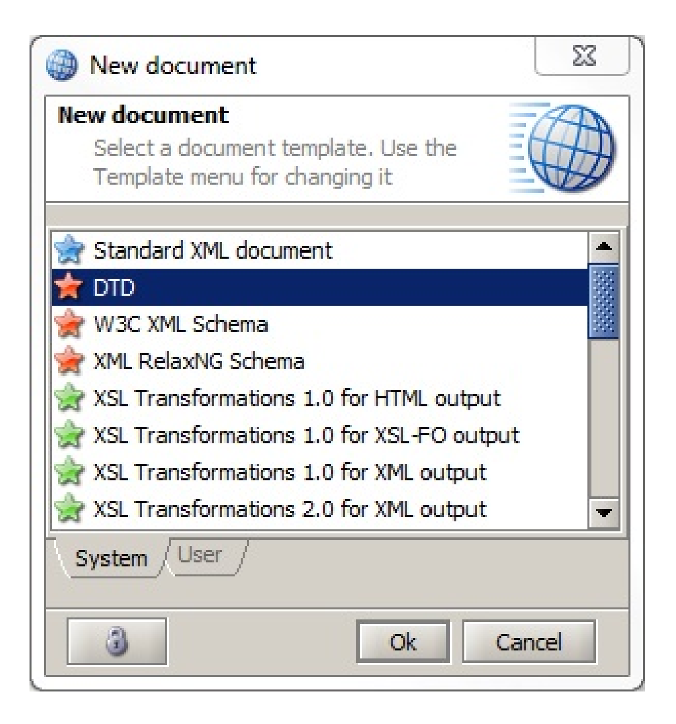
## Création du document DTD

Pour créer un nouveau document, vous pouvez cliquer sur l'icône suivante :



sélectionner dans la barre de menu `File`puis `New`ou encore utiliser le raccourci clavier Ctrl + N.

Dans la liste qui s'affiche, sélectionnez `DTD` (voir la figure suivante).




Votre document DTD n'est normalement pas vierge. Voici ce que vous devriez avoir :

**Code : XML**

```
<!-- DTD created at Wed Sep 12 14:49:47 CEST 2012 with EditiX.
Please insert an encoding attribute header for converting any DTD -
->

<!ELEMENT tag (#PCDATA)>
<!ATTLIST tag attribute CDATA #REQUIRED>
```

Replacez le contenu par notre véritable DTD :

**Code : XML**

```
<!ELEMENT boutique (telephone*)>
<!ELEMENT telephone (marque, modele)>
<!ELEMENT marque (#PCDATA)>
<!ELEMENT modele (#PCDATA)>
```

Enregistrez ensuite votre document avec le nom **boutique.dtd** au même endroit que votre document XML.

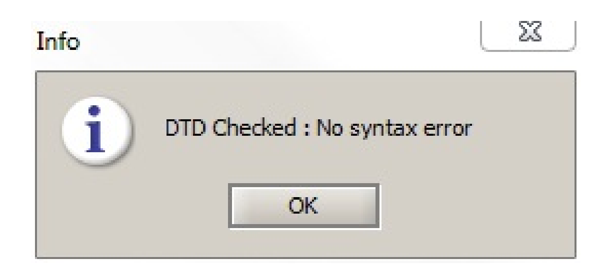
## Vérification de la DTD

Vous pouvez vérifier que votre DTD n'a pas d'erreur de syntaxe en cliquant sur l'icône suivante



ou bien en sélectionnant dans la barre de menu `DTD/Schema` puis `Check this DTD` ou encore en utilisant le raccourci clavier Ctrl + K.

Vous devriez normalement avoir un message d'information (voir la figure suivante).

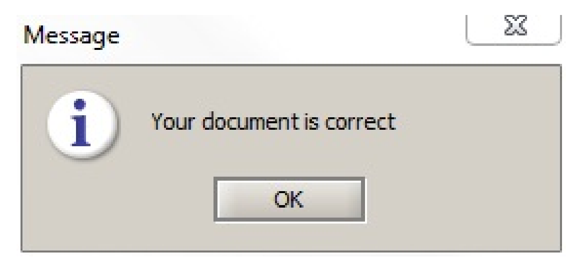



## Vérification du document XML

Il est maintenant temps de vérifier que le document XML est **valide** !

Pour ce faire, sélectionnez dans la barre de menu `XML` puis `Check this document` ou encore en utilisant le raccourci clavier Ctrl + K.

Le message suivant doit normalement s'afficher (voir la figure suivante).




Nous venons donc de voir qu'il est possible d'écrire une DTD au sein même du fichier XML qu'elle définit. Cependant, nous privilégierons l'écriture de ces définitions dans un second fichier portant l'extension **.dtd**.

#  TP : définition DTD d'un répertoire

Votre apprentissage des DTD arrive donc à son terme et rien ne vaut un TP pour le conclure !

Pour ce TP, je vous propose de réaliser la définition DTD d'un répertoire téléphonique. Si en soit le sujet n'est pas extrêmement compliqué, il a le mérite de vous faire mettre en pratique toutes les notions vues jusque là dans cette seconde partie.

## L'énoncé

Le but de ce TP est de créer la DTD du répertoire élaboré dans le premier TP.

Pour rappel, voici les informations que l'on souhaite connaître pour chaque personne :

- son sexe (homme ou femme) ;
- son nom ;
- son prénom ;
- son adresse ;
- un ou plusieurs numéros de téléphone (téléphone portable, fixe, bureau, etc.) ;
- une ou plusieurs adresses e-mail (adresse personnelle, professionnelle, etc.).

Voici le document XML que nous avions construit :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <adresse>
            <numero>7</numero>
            <voie type="impasse">impasse du chemin</voie>
            <codePostal>75015</codePostal>
            <ville>PARIS</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
        <emails>
            <email type="personnel">john.doe@wanadoo.fr</email>
            <email type="professionnel">john.doe@societe.com</email>
        </emails>
    </personne>

    <!-- Marie POPPINS -->
    <personne sexe="feminin">
        <nom>POPPINS</nom>
        <prenom>Marie</prenom>
        <adresse>
            <numero>28</numero>
            <voie type="avenue">avenue de la république</voie>
            <codePostal>13005</codePostal>
            <ville>MARSEILLE</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="bureau">04 05 06 07 08</telephone>
        </telephones>
        <emails>
            <email type="professionnel">contact@poppins.fr</email>
        </emails>
```

```
        </personne>
    </repertoire>
```

Une dernière consigne : la DTD doit être une DTD externe !

## Une solution

Une fois de plus, je vous fais part de ma solution !

Le fichier XML avec le DOCTYPE :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<!DOCTYPE repertoire SYSTEM "repertoire.dtd">

<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <adresse>
            <numero>7</numero>
            <voie type="impasse">impasse du chemin</voie>
            <codePostal>75015</codePostal>
            <ville>PARIS</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
        <emails>
            <email type="personnel">john.doe@wanadoo.fr</email>
            <email type="professionnel">john.doe@societe.com</email>
        </emails>
    </personne>

    <!-- Marie POPPINS -->
    <personne sexe="feminin">
        <nom>POPPINS</nom>
        <prenom>Marie</prenom>
        <adresse>
            <numero>28</numero>
            <voie type="avenue">avenue de la république</voie>
            <codePostal>13005</codePostal>
            <ville>MARSEILLE</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="professionnel">04 05 06 07
08</telephone>
        </telephones>
        <emails>
            <email type="professionnel">contact@poppins.fr</email>
        </emails>
    </personne>
</repertoire>
```

Le fichier DTD :

**Code : XML**

```
<!-- Racine -->
<!ELEMENT repertoire (personne*)>

<!-- Personne -->
<!ELEMENT personne (nom, prenom, adresse, telephones, emails)>
<!ATTLIST personne sexe (masculin | feminin) #REQUIRED>

<!-- Nom et prénom -->
<!ELEMENT nom (#PCDATA)>
<!ELEMENT prenom (#PCDATA)>

<!-- Bloc adresse -->
<!ELEMENT adresse (numero, voie, codePostal, ville, pays)>
<!ELEMENT numero (#PCDATA)>

<!ELEMENT voie (#PCDATA)>
<!ATTLIST voie type CDATA #REQUIRED>

<!ELEMENT codePostal (#PCDATA)>
<!ELEMENT ville (#PCDATA)>
<!ELEMENT pays (#PCDATA)>

<!-- Bloc téléphone -->
<!ELEMENT telephones (telephone+)>
<!ELEMENT telephone (#PCDATA)>
<!ATTLIST telephone type CDATA #REQUIRED>

<!-- Bloc email -->
<!ELEMENT emails (email+)>
<!ELEMENT email (#PCDATA)>
<!ATTLIST email type CDATA #REQUIRED>
```

## Un bref commentaire

Dans cette solution, je suis allé au plus simple en indiquant que pour les types de téléphones, d'e-mails et de voies, j'accepte toutes les chaînes de caractères. Libre à vous de créer de nouvelles règles si vous souhaitez que par exemple le choix du type de la voie ne soit possible qu'entre rue, avenue, impasse, etc.

#  Schéma XML : introduction

Pour écrire les définitions de vos documents XML, vous pouvez utiliser les DTD ou les schémas XML. Puisque nous avons déjà vu les DTD, il est temps de s'attaquer aux XML Schema !

## Les défauts des DTD

Peut-être l'avez vous remarqué dans les précédents chapitres, mais les DTD ont quelques défauts. Si vous ne les avez pas remarqué, je vous propose de revenir dessus ensemble.

### Un nouveau format

Tout d'abord, les DTD ne sont pas au format XML. Nous avons dû apprendre un nouveau langage avec sa propre syntaxe et ses propres règles.

La principale conséquence est que pour exploiter une DTD, nous allons être obligé d'utiliser un outil différent de celui qui exploite un fichier XML. Il est vrai que dans notre cas, nous avons utilisé le même outil, à savoir EditiX, mais vos futurs programmes, logiciels ou applications mobiles devront forcément exploiter la DTD et le fichier XML différemment, via par exemple une API différente.

### Le typage de données

Le second défaut que l'on retiendra dans ce cours est que les DTD ne permettent pas de typer des données. Comme vous avez pu le voir, on se contente d'indiquer qu'une balise contient des données, mais impossible de préciser si l'on souhaite que ça soit un nombre entier, un nombre décimal, une date, une chaîne de caractères, etc.

## Les apports des schémas XML

C'est pour paliers les défauts des DTD que les Schémas XML ont été créés. S'ils proposent au minimum les mêmes fonctionnalités que les DTD, ils en proposent également de nouvelles. En voici quelques unes en vrac.

### Le typage des données

Les Schémas XML permettent tout d'abord de typer les données. Nous verrons également dans la suite de ce tutoriel, qu'il est possible d'aller plus loin en créant nos propres types de données.

### Les contraintes

Nous verrons également que les Schémas XML permettent d'être beaucoup plus précis que les DTD lors de l'écriture des différentes contraintes qui régissent un document XML.

### Des définitions XML

Un des principaux avantages des Schémas XML est qu'ils s'écrivent grâce au XML. Ainsi, pour exploiter un document XML et le Schéma qui lui est associé, vous n'avez en théorie plus besoin de plusieurs outils. Dorénavant un seul suffit !

## Structure d'un schéma XML

Maintenant que vous en savez un peu plus sur les Schémas XML, je vous propose de voir les bases qui permettent de définir un schéma XML.

### L'extension du fichier

Comme pour les DTD, nous prendrons l'habitude de séparer les données formatées avec XML et le Schéma XML associé dans 2 fichiers distincts.

Bien que c'est les Schémas XML soient écrits avec un langage de type XML, le fichier n'a pas cette extension. Un fichier dans lequel est écrit un Schéma XML porte l'extension "**.xsd**".

### Le prologue

Puisque c'est le XML qui est utilisé, il ne faut déroger à la règle du prologue.

Ainsi, la première ligne d'un Schémas XML est :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Je ne détaille pas ici les différents éléments du prologue puisque je l'ai déjà fait lors de la première partie dans le chapitre traitant de la structure d'un document XML. Si vous avez des doutes, je vous encourage vivement à relire cette partie !

### Le corps

Comme pour un fichier XML classique, le corps d'un Schéma XML est constitué d'un ensemble de balises dont nous verrons le rôle dans les prochains chapitres.

Cependant, une chose ne change pas : la présence d'un **élément racine**, c'est-à-dire la présence d'une balise qui contient toutes les autres. Mais contrairement à un fichier XML, son nom nous est imposé.

**Code : XML**

```
<!-- Prologue -->
<?xml version="1.0" encoding="UTF-8" ?>

<!-- Elément racine -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

</xsd:schema>
```

Comme vous pouvez le voir dans le code précédent, l'élément racine est `<xsd:schema />`.

Si l'on regarde de plus prêt, on remarque la présence de l'attribut **xmlns:xsd**. **xmlns** nous permet de déclarer un espace de nom. Si ce vocabulaire ne vous parle pas, je vous encourage à lire le chapitre dédié à cette notion en annexe de ce tutoriel.

Via la déclaration de cet espace de nom, tous les éléments doivent commencer par **xsd:**.

## Référencer un schéma XML

Le référencement d'un schéma XML, ce fait au niveau de l'élément racine du fichier XML grâce à l'utilisation de 2 attributs.

### L'espace de noms

**Code : XML**

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

### La location

Le second attribut nous permet d'indiquer à notre fichier XML où se situe le fichier contenant le Schéma XML.

2 possibilités s'offrent alors à nous : les schémas XML qui décrivent un espace de noms et ceux qui ne décrivent pas un espace de noms.

### *Schéma XML décrivant un espace de noms*

**Code : XML**

```
xsi:schemaLocation="chemin_vers_fichier.xsd">
```

### *Schéma XML ne décrivant pas un espace de noms*

Dans les prochains chapitre, c'est ce type de Schéma XML que nous allons utiliser.

On utilisera alors la syntaxe suivante :

**Code : XML**

```
xsi:noNamespaceSchemaLocation="chemin_vers_fichier.xsd">
```

## Pour résumer

Pour résumer, voici ce à quoi nos fichiers XML ressembleront :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>

<racine xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="chemin_vers_fichier.xsd">

</racine>
```

Maintenant que vous savez un peu plus sur les motivations des schémas XML, je vous propose de rentrer dans le vif du sujet !

#  Schéma XML : les éléments simples

Dans les schémas XML, on distinguera 2 types d'éléments :

- les éléments simples ;
- les éléments complexes.

Ici c'est les éléments simples que nous allons étudier. Nous reviendrons sur les éléments complexes dans les prochains chapitres.

## Les éléments simples

## Définition

Un élément simple est un élément qui ne contient qu'une valeur dont le type est dit **simple**. Il ne contient pas d'autres éléments.

Un élément simple peut donc être une balise qui ne contient aucun attribut et dans laquelle aucune autre balise n'est imbriquée. Un attribut d'une balise peut également être considéré comme un élément simple. En effet, la valeur d'un attribut est un type simple.

Nous verrons la liste complète des types simples un peu plus loin dans ce tutoriel, mais je peux déjà vous citer quelques exemple afin de tenter d'éclaircir les choses. Un type simple, c'est par exemple un chiffre, une date ou encore une chaîne de caractères.

## Quelques exemples

Prenons quelques exemples de fichiers XML, et regardons ensemble s'ils peuvent être considérés comme des types simples :

**Code : XML**

```
<!-- Ne contient ni attribut ni aucun autre élément => élément
simple -->
<nom>ROBERT</nom>

<!-- Contient un attribut => n'est pas un élément simple -->
<!-- Cependant l'attribut "sexe" est un élément simple -->
<personne sexe="masculin>Robert DUPONT</personne>

<!-- La balise personne contient d'autres éléments (les balises nom
et prénom) => n'est pas un élément simple -->
<personne>
    <!-- Ne contient ni attribut ni aucun autre élément => élément
simple -->
    <nom>DUPONT</nom>

    <!-- Ne contient ni attribut ni aucun autre élément => élément
simple -->
    <prenom>Robert</prenom>
</personne>
```

J'espère que ces exemples vous permettent de mieux comprendre ce qu'est un élément simple.

## Déclarer une balise comme un élément simple

Si vous souhaitez déclarer une balise en tant qu'élément simple, c'est le mot clef **element** qu'il faut utiliser. N'oubliez pas de précéder son utilisation par **xsd:**

Cette balise prend 2 attributs : un nom et un type.

**Code : XML**

```
<xsd:element name="mon_nom" type="xsd:mon_type" />
```

Voyons tout de suite un exemple. Soit les éléments simples suivants :

**Code : XML**

```
<nom>DUPONT</nom>
<prenom>Robert</prenom>
<age>38</age>
```

Au sein d'un Schéma XML, les éléments précédents seront déclarés de la sorte :

**Code : XML**

```
<xsd:element name="nom" type="xsd:string" />
<xsd:element name="prenom" type="xsd:string" />
<xsd:element name="age" type="xsd:int" />
```


C'est quoi ce int et ces strings ?

**String** est utilisé pour qualifier **une chaîne de caractères** et **int** est utilisé pour qualifier **un nombre entier**.

La liste complète des types des types qu'il est possible d'utiliser est fournie un peu plus loin dans ce tutoriel. 

## Valeur par défaut et valeur inchangeable

Avant que l'on regarde ensemble la liste des types, j'aimerais revenir sur 2 concepts qui sont les **valeurs par défaut** et les **valeurs inchangeables**.

### *Valeur par défaut*

Comme c'était déjà le cas dans les DTD, il est tout à fait possible d'indiquer dans les Schémas XML qu'un élément a une valeur par défaut. Pour rappel, la valeur par défaut est la valeur que va prendre automatiquement un élément si aucune valeur n'est indiquée au niveau du fichier XML.

Pour indiquer une valeur par défaut, c'est l'attribut **default** qui est utilisé au niveau de la balise `<element />` du Schéma XML. Par exemple, si je souhaite indiquer qu'à défaut d'être renseigné, le prénom d'une personne est Robert, je vais écrire la règle suivante :

**Code : XML**

```
<xsd:element name="prenom" type="xsd:string" default="Robert" />
```

Voici alors quelques exemple de documents XML possibles :

**Code : XML**

```
<!-- valide -->
<prenom>Jean</prenom>
```

```
<!-- valide -->
<prenom>Marie</prenom>

<!-- valide -->
<!-- la balise prenom vaut "Robert" -->
<prenom />
```

### *Valeur constante*

S'il est possible d'indiquer une valeur par défaut, il est également possible d'imposer une valeur. Cette valeur inchangeable est appelée **constante**.

Pour indiquer une valeur constante, c'est l'attribut **fixed** qui est utilisé au niveau de la balise `<element />` du Schéma XML. Par exemple, si je souhaite obliger toute les personnes de mon document XML à porter le prénom Robert, voici la règle à écrire :

**Code : XML**

```xml
<xsd:element name="prenom" type="xsd:string" fixed="Robert" />
```

Voyons alors la validité des lignes XML suivantes :

**Code : XML**

```xml
<!-- valide -->
<prenom>Robert</prenom>

<!-- invalide -->
<prenom>Marie</prenom>

<!-- invalide -->
<prenom/>
```

## Les attributs

Comme je vous le disais un peu plus tôt, dans un Schéma XML, tous les attributs d'une balise XML sont considérés comme des éléments simples. En effet, ils ne peuvent prendre comme valeur qu'un type simple, c'est-à-dire un nombre, une chaîne de caractère, une date, etc.

## Déclarer un attribut

Bien qu'un attribut soit un élément simple, nous n'allons pas utiliser le mot clef **element** pour déclarer un attribut. C'est le mot **attribut** qui est utilisé. Encore une fois, n'oubliez pas de précéder son utilisation par **xsd:**

Cette balise prend 2 attributs : un nom et un type.

**Code : XML**

```xml
<xsd:attribut name="mon_nom" type="xsd:mon_type" />
```

Prenons par exemple la ligne XML suivante :

**Code : XML**

```xml
<personne sexe="masculin">Robert DUPONT</personne>
```

Je ne vais pas détailler ici comment déclarer la balise. En effet, puisqu'elle contient un attribut, c'est ce qu'on appelle un élément complexe et nous verrons comment faire un peu plus tard. Cependant, voici comme déclarer l'attribut dans notre Schéma XML :

**Code : XML**

```
<xsd:attribut name="sexe" type="xsd:string" />
```

## Valeur par défaut, obligatoire et inchangeable

### *Valeur par défaut*

Comme c'était déjà le cas dans les DTD, il est tout à fait possible d'indiquer dans les Schémas XML qu'un attribut a une valeur par défaut. Pour rappel, la valeur par défaut est la valeur prise automatiquement par un attribut si aucune valeur n'est indiquée au niveau du fichier XML.

Pour indiquer une valeur par défaut, c'est l'attribut **default** qui est utilisé au niveau de la balise `<attribut />` du Schéma XML. Par exemple, si je souhaite indiquer qu'à défaut d'être renseigné, le prénom d'une personne est Robert, je vais écrire la règle suivante :

**Code : XML**

```
<xsd:attribut name="prenom" type="xsd:string" default="Robert" />
```

### *Valeur constante*

S'il est possible d'indiquer une valeur par défaut, il est également possible d'imposer une valeur. Cette valeur inchangeable est appelée **constante**.

Pour indiquer une valeur constante, c'est l'attribut **fixed** qui est utilisé au niveau de la balise `<attribut />` du Schéma XML. Par exemple, si je souhaite obliger toutes les personnes de mon document XML à porter le prénom Robert, voici la règle à écrire :

**Code : XML**

```
<xsd:attribut name="prenom" type="xsd:string" fixed="Robert" />
```

### *Attribut obligatoire*

Tel que nous les déclarons depuis le début de ce chapitre, les attributs sont par défaut optionnels.

Pour indiquer qu'un attribut est obligatoire, nous devons renseigner la propriété **use** à laquelle nous affectons la valeur **required**. Par exemple, si je souhaite obliger l'utilisation de l'attribut **prenom**, voici la règle à écrire :

**Code : XML**

```
<xsd:attribut name="prenom" type="xsd:string" use="required" />
```

Maintenant que vous savez ce qu'est un élément simple, passons tout de suite aux éléments complexes !

#  Schéma XML : les types simples

Voici donc la liste des types simples. Nous pouvons les regrouper en 4 grandes catégories :

- les chaînes de caractères ;
- les dates ;
- les nombres ;
- les types divers.

Commençons tout de suite avec les chaînes de caractères.

## Les types chaînes de caractères

## Le tableau récapitulatif

| Type | Description | Commentaire |
|---|---|---|
| string | représente une chaîne de caractères | attention aux caractères spéciaux |
| normalizedString | représente une chaîne de caractères normalisée | basé sur le type string |
| token | représente une chaîne de caractères normalisée sans espace au début et à la fin | basé sur le type normalizedString |
| language | représente le code d'une langue | basé sur le type token |
| NMTOKEN | représente une chaîne de caractère "simple" | basé sur le type token applicable uniquement aux attributs |
| NMTOKENS | représente une liste de NMTOKEN | applicable uniquement aux attributs |
| Name | représente un nom XML | basé sur le type token |
| NCName | représente un nom XML sans le caractère **:** | basé sur le type Name |
| ID | représente un identifiant unique | basé sur le type NCName applicable uniquement aux attributs |
| IDREF | référence à un identifiant | basé sur le type NCName applicable uniquement aux attributs |
| IDREFS | référence une liste d'identifiants | applicable uniquement aux attributs |
| ENTITY | représente une entité d'un document DTD | basé sur le type NCName applicable uniquement aux attributs |
| ENTITIES | représente une liste d'entités | applicable uniquement aux attributs |

## Plus en détails

### *Le type string*

Le type **string** est l'un des premiers types que nous ayons vu ensemble. Il représente une chaîne de caractères et peut donc contenir un peu tout et n'importe quoi. Il est cependant important de noter que certains caractères spéciaux comme le **&** doivent être écrits avec leur notation HTML.

Une liste des caractères spéciaux et de leur notation HTML est disponible ici.

Bien que nous ayons déjà vu plusieurs exemples ensemble, je vous propose d'en voir un nouveau. Soit la règle de Schéma XML suivante :

**Code : XML**

```
<xsd:element name="string" type="xsd:string" />
```

Les différentes lignes XML ci-dessous sont alors valides :

**Code : XML**

```
<string>France</string>

<string>Site du zéro !</string>

<string>&amp;</string>
```

### *Le type normalizedString*

Le type **normalizedString** est basé sur le type **string** et représente une chaîne de caractères normalisée, c'est-à-dire une chaîne de caractères qui peut contenir tout et n'importe quoi à l'exception de tabulations, de sauts de ligne et de retours chariot. Dans la pratique, il n'est pas interdit de les écrire, mais ils seront automatiquement remplacés par des espaces.

Puisque le type **normalizedString** est basé sur le type **string**, toutes les règles du type string s'applique également au type **normalizedString**. Ainsi, les caractères spéciaux comme le **&** doivent être écrits avec leur notation HTML.

Je ne le préciserai pas à chaque fois, mais cette règle est toujours vraie. Un type hérite toujours de toutes les règles du type sur lequel il se base.

### *Le type token*

Le type **token** est basé sur le type **normalizedString** et représente une chaîne de caractères normalisée sans espace au début ni à la fin. Une nouvelle fois, dans la pratique, il n'est pas interdit de les écrire. Les espaces présents au début et à la fin seront automatiquement supprimés.

### *Le type language*

Le type **language** est basé sur le type **token** et représente, comme son nom le laisse deviner, une langue. Cette langue doit être identifiée par 2 lettre (selon la norme ISO 639 dont la liste est disponible par sur wikipedia). Ces 2 caractères peuvent éventuellement être suivi d'un code pays (selon la norme ISO 3166 dont la liste est une nouvelle fois disponible sur wikipedia).

Considérons la règle suivante :

**Code : XML**

```
<xsd:element name="langue" type="xsd:language" />
```

Les différentes lignes XML ci-dessous sont alors valides :

**Code : XML**

```
<langue>fr</langue>

<langue>en</langue>
```

```xml
<langue>en-GB</langue>
<langue>en-US</langue>
```

### Le type NMTOKEN

Le type **NMTOKEN** est basé sur le type **token** et représente une chaîne de caractères "simple", c'est-à-dire une chaîne de caractères sans espace qui ne contient que les symboles suivants :

- des lettres ;
- des chiffres ;
- les caractères spéciaux **. - _** et **:**

Si la chaîne de caractères contient des espaces au début ou à la fin, ils seront automatiquement supprimés.

Afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type **NMTOKEN** que pour un attribut.

### Le type NMTOKENS

Le type **NMTOKENS** représente une liste de **NMTOKEN** séparés par un espace. Une nouvelle fois, afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type **NMTOKENS** que pour un attribut.

Soit la règle suivante issue d'un Schéma XML :

**Code : XML**

```xml
<xsd:attribut name="liste" type="xsd:NMTOKENS" />
```

Les différentes lignes XML ci-dessous sont alors valides :

**Code : XML**

```xml
<balise list="A:1_B C-2.">contenu de la balise</balise>
<balise list="AZERTY 123456 QSDFGH">contenu de la balise</balise>
```

### Le type Name

Le type **Name** est basé sur le type **token** et représente un nom XML, c'est-à-dire une chaîne de caractères sans espace qui ne contient que les symboles suivants :

- des lettres ;
- des chiffres ;
- les caractères spéciaux **. - _** et **:**

La différence avec le type **NMTOKEN** est qu'une chaîne de caractères de type **Name** doit obligatoirement commencée par une lettre, ou l'un des 2 caractères spéciaux suivants : **_** et **:**

### Le type NCName

Le type **NCName** est basé sur le type **Name**. Il hérite donc de toutes les règles du type **Name** auxquelles une nouvelle règles doit être ajoutée : le type **NCName** ne peut pas contenir le caractère spécial **:**

### Le type ID

Le type **ID** est basé sur le type **NCName**, il hérite donc de toutes les règles de ce type. Comme son nom le laisse deviner, un **ID** représente un identifiant. Il doit donc contenir des valeurs uniques. A ce titre, il est impossible de lui définir une valeur fixe ou par défaut.

Comme pour d'autres types vu précédemment, un **ID** ne doit être utilisé qu'avec les attributs afin d'assurer une compatibilité entre les Schémas XML et les DTD.

### Le type IDREF

Le type **IDREF** fait référence à un **ID** existant dans le document XML. Tout comme le type **ID**, il est basé sur le type **NCName** et hérite donc de toutes les règles de ce type. Puisque le type **ID** n'est utilisable qu'avec des attributs, il en est naturellement de même pour le type **IDREF**.

### Le type IDREFS

Si le type **NMTOKENS** représente une liste de **NMTOKEN** séparés par un espace, le type **IDREFS** représente lui une liste de **IDREF** séparés par un espace.

Afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type **IDREFS** que pour un attribut.

Illustrons son utilisation avec un exemple :

**Code : XML**

```xml
<xsd:attribut name="enfants" type="xsd:IDREFS" />
```

**Code : XML**

```xml
<personne num="P1">Paul</personne>
<personne num="P2">Marie</personne>

<personne enfants="P1 P2">Jeanne</personne
```

### Le type ENTITY

Le type **ENTITY** permet de faire référence à une entité le plus souvent non XML et déclaré dans des fichiers DTD. Ce type est basé sur le type **NCName**, il hérite donc de toutes ses règles.

Une nouvelle fois, afin d'assurer une compatibilité entre les Schémas XML et les DTD, il convient de n'utiliser le type **ENTITY** que pour un attribut.

Une nouvelle fois, je vous propose d'illustrer son utilisation par un exemple :

**Code : XML**

```xml
<xsd:attribut name="marque" type="xsd:ENTITY" />
```

**Code : XML**

```xml
<!ENTITY samsung "Samsung">
```

```
<!ENTITY apple "Apple">

<telephone marque="apple">iPhone</personne>
<telephone marque="samsung">Galaxy SII</personne>
```

### *Le type ENTITIES*

Finalement, le dernier type que nous verrons dans cette catégorie est le type **ENTITIES**. Il permet de faire référence à une liste d'**ENTITY** séparés par un espace.

Puisque c'était déjà le cas pour le type **ENTITY**, le type **ENTITIES** n'échappe pas à la règle et ne doit être utilisé qu'avec un attribut.

## Les types dates

## Le tableau récapitulatif

| Type | Description |
|---|---|
| duration | représente une durée |
| date | représente une date |
| time | représente une heure |
| dateTime | représente une date et un temps |
| gYear | représente une année |
| gYearMonth | représente une année et un mois |
| gMonth | représente un mois |
| gMonthDay | représente un mois et un jour |
| gDay | représente un jour |

## Plus en détails

### *Le type duration*

Le type **duration**, comme son nom le laisse deviner, représente une durée. Cette durée s'exprime en nombre d'années, de mois, de jours, d'heures, de minutes et de secondes selon une expression qui n'est pas des plus simples à savoir PnYnMnDTnHnMnS.

Je vous propose de la décortiquer :

- P marque le début de l'expression ;
- nY représente le nombre d'années (year) où n est un nombre entier ;
- nM représente le nombre de mois (month) où n est un nombre entier ;
- nD représente le nombre de jours (day) où n est un nombre entier ;
- T permet de séparer la partie date de l'expression de sa partie heure ;
- nH représente le nombre d'heures (hour) où n est un nombre entier ;
- nM représente le nombre de minutes (minute) où n est un nombre entier ;
- nS représente le nombre de secondes (second) où n est un nombre entier ou décimal ;

L'expression peut-être précédé du signe - dans le cas où l'on souhaite exprimer une durée négative. Bien évidemment, tous les champs ne doivent pas forcément être renseignés. Ainsi, il est possible de ne renseigner que les heures, les minutes, etc. Dans le cas où l'expression n'exprime qu'une date, le symbole T ne doit plus figurer.

Je vous accorde que toutes ces règles ne sont pas facile à assimiler, c'est pourquoi je vous propose de voir quelques exemples :

**Code : XML**

```
<xsd:element name="duree" type="xsd:duration" />
```

**Code : XML**

```
<!-- 42 ans et 6 minutes -->
<duree>P42YT6M</duree>

<!-- -2 heures -->
<duree>-PT2H</duree>

<!-- 2 jours -->
<duree>P2D</duree>

<!-- 10.5 secondes -->
<duree>PT10.5S</duree>
```

### *Le type date*

Le type **date** permet d'exprimer une date. A l'image du type **duration**, une date s'exprime selon une expression bien spécifique à savoir YYYY-MM-DD.

Une nouvelle fois, je vous propose de décortiquer tout ça :

- YYYY représente l'année (year) sur 4 chiffres ou plus ;
- MM représente le mois (month) sur 2 chiffres ;
- DD représente le jour (day) également sur 2 chiffres ;

Dans le cas où l'on souhaite exprimer une date avant Jesus-Christ, un signe **-** peut-être placé devant l'expression.

Voyons ensemble quelques exemples :

**Code : XML**

```
<xsd:element name="madate" type="xsd:date" />
```

**Code : XML**

```
<!-- 13 janvier 1924 -->
<madate>1924-01-13</madate>

<!-- 12 décembre 34 avant JC -->
<madate>-0034-12-12</madate>

<!-- 4 novembre 12405 -->
<madate>12405-11-04</madate>
```

### *Le type time*

Le type **time** permet d'exprimer une heure. Encore une fois, une expression bien spécifique doit être respectée : hh:mm:ss.

Pour ne rien changer, décortiquons ensemble cette expression :

- hh représente les heures (hour) sur 2 chiffres ;
- mm représente les minutes (minute) sur 2 chiffres ;

- ss représente les secondes (second) sur 2 chiffres entiers ou à virgule ;

Voici quelques exemples :

**Code : XML**

```
<xsd:element name="monheure" type="xsd:time" />
```

**Code : XML**

```
<!-- 10 heures et 24 minutes -->
<monheure>10:24:00</monheure>

<!-- 2,5 secondes -->
<monheure>00:00:02.5</monheure>
```

### Le type dateTime

Le type **dateTime** peut-être considéré comme un mélange entre le type **date** et le type **time**. Ce nouveau type permet donc de représenter une date ET une heure. Une nouvelle fois, une expression particulière doit être respectée : YYYY-MM-DDThh:mm:ss.

Je ne vais pas spécifiquement revenir sur cette expression. En effet, comme vous pouvez le constater, il s'agit des expressions du type **date** et du type **time** séparées par la lettre **T**. Je vous laisse vous référer aux types **date** et **time** pour les règles à appliquer.

### Le type gYear

Le type **gYear** représente une année sur 4 chiffres ou plus. Dans le cas où l'on souhaite exprimer une année avant Jesus-Christ, un signe **-** peut-être placé devant l'expression.

### Le type gYearMonth

Le type **gYearMonth** représente une année et un mois. Comme pour tous les types que nous venons de voir dans ce chapitre, le type **gYearMonth** doit respecter une expression particulière : YYYY-MM.

Vous l'aurez compris, les règles sont toujours les mêmes. Je vous laisse donc vous reporter au paragraphe traitant du type date pour plus d'information.

### Le type gMonth

Le type **gMonth** représente un mois sur 2 chiffres précédés du symbole **--**.

Non, ce n'est pas une erreur de frappe, le symbole est bien **--**. Voyons un exemple :

**Code : XML**

```
<xsd:element name="mois" type="xsd:gMonth" />
```

**Code : XML**

```
<!-- mars -->
```

```xml
<mois>--03</mois>

<!-- décembre -->
<mois>--12</mois>
```

### *Le type gMonthDay*

Le type **gMonthDay** représente un mois et un jour. Une nouvelle fois, une expression particulière doit être utilisée afin d'exprimer ce nouveau type : --MM-DD.

Une nouvelle fois, les règles sont les mêmes que nous avons déjà utilisé précédemment notamment pour le type **date** et me type **gYearMonth**.

### *Le type gDay*

Finalement, nous allons terminer ce chapitre avec le type **gDay** qui représente un jour sur 2 chiffres précédés du symbole **---**.

Afin de terminer ce chapitre en beauté, voici quelques exemples :

**Code : XML**

```xml
<xsd:element name="journee" type="xsd:gDay" />
```

**Code : XML**

```xml
<!-- le troisième jour du mois -->
<journee>---03</journee>

<!-- le douzième jour du mois -->
<journee>---12</journee>
```

## Les types numériques

## Le tableau récapitulatif

| Type | Description | Commentaire |
|---|---|---|
| float | représente un nombre flottant sur 32 bits conforme à la norme IEEE 754 | |
| double | représente un nombre flottant sur 64 bits conforme à la norme IEEE 754 | |
| decimal | représente une nombre décimal | |
| integer | représente un nombre entier | basé sur le type decimal |
| long | représente un nombre entier | basé sur le type integer |
| int | représente un nombre entier | basé sur le type long |
| short | représente un nombre entier | basé sur le type int |
| byte | représente un nombre entier | basé sur le type short |
| nonPositiveInteger | représente un nombre entier non positif | basé sur le type integer |
| negativeInteger | représente un nombre entier négatif | basé sur le type nonPositiveInteger |

| | | |
|---|---|---|
| nonNegativeInteger | représente un nombre entier non négatif | basé sur le type integer |
| positiveInteger | représente un nombre entier positif | basé sur le type nonNegativeInteger |
| unsignedLong | représente un nombre entier positif | basé sur le type nonNegativeInteger |
| unsignedInt | représente un nombre entier positif | basé sur le type unsignedLong |
| unsignedShort | représente un nombre entier positif | basé sur le type unsignedInt |
| unsignedByte | représente un nombre entier positif | basé sur le type unsignedShort |


Comme bien souvent en informatique, il convient d'écrire d'écrire les nombres décimaux avec un point et non une virgule. Par exemple 4.2, 5.23, etc.

## Plus en détails

### *Le type float*

Comme vous avez déjà pu le lire dans le tableau récapitulatif, le type **float** représente un nombre flottant sur 32 bits et conforme à la norme IEEE 754. Je suis parfaitement conscient que cette définition est incompréhensible pour la plupart des gens, c'est pourquoi nous allons grandement la simplifier.

Le type **float** à été emprunté aux langages de programmation comme le langage C et est encore aujourd'hui utilisé dans des langages plus récents comme Java ou C#. Il représente un nombre flottant, c'est-à-dire un nombre entier ou décimal, se trouvant entre les valeurs 3.4x10^-38 et 3.4x10^38.

A cette plage de valeurs, 3 autres peuvent être ajoutées :

- -INF pour moins l'infini ;
- +INF pour plus l'infini ;
- NaN pour Not a Number, c'est-à-dire pour désigner une valeur non numérique.

Il est tout à fait possible d'écrire un nombre de type **float** avec des exposants. Il convient alors d'utiliser la notation **E** ou **e**.

Pour mieux comprendre toutes ces règles, je vous propose de regarder ensemble quelques exemples :

**Code : XML**

```xml
<xsd:element name="nombre" type="xsd:float" />
```

**Code : XML**

```xml
<nombre>42</nombre>

<nombre>-42.25</nombre>

<nombre>3E4</nombre>

<nombre>10e-5</nombre>
```

### *Le type double*

Le type **double** est très proche du type **float**, si ce n'est qu'il représente un nombre flottant sur 64 bits et conforme à la norme IEEE 754 au lieu des 32 bits du type **float**. Concrètement, cette différence se traduit par le fait qu'un nombre de type **double** se trouvant

entre les valeurs 1.7x10^-308 et 1.7x10^308.

Comme pour le type **float**, les 3 valeurs suivantes peuvent être ajoutées à la liste :

- -INF pour moins l'infini ;
- +INF pour plus l'infini ;
- NaN pour Not a Number, c'est-à-dire pour désigner une valeur non numérique.

On retrouve également la règle de l'exposant. Je vous laisse vous référer à la définition du type **float** pour plus de détails.

### *Le type decimal*

Comme son nom le laisse deviner, le type **decimal** représente un nombre décimal, c'est-à-dire un nombre qui peut-être entier ou à virgule. Ce nombre peut-être positif ou négatif et donc être précédé du symbole **+** ou **-**. Dans le cas d'un nombre où la partie entière est égale à zéro, il n'est pas obligatoire de l'écrire.

Voyons tout de suite quelques exemples afin d'illustrer cette définition :

**Code : XML**

```
<xsd:element name="nombre" type="xsd:decimal" />
```

**Code : XML**

```
<nombre>42</nombre>
<nombre>-42.25</nombre>
<nombre>+.42</nombre>
<nombre>00042.420000</nombre>
```

### *Le type integer*

Le type **integer** est basé sur le type **decimal** et représente un nombre entier, c'est-à-dire un nombre sans virgule. Comme pour le type **décimal**, un nombre de type **integer** peut être précédé par le symbole **+** ou **-**.

### *Le type long*

Le type **long** est basé sur le type **integer** si ce n'est qu'un nombre de type **long** doit forcément être compris entre les valeurs -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807.

### *Le type int*

Le type **int** est basé sur le type **long** si ce n'est qu'un nombre de type **int** doit forcément être compris entre les valeurs -2 147 483 648 et 2 147 483 647.

### *Le type short*

Le type **short** est basé sur le type **int** si ce n'est qu'un nombre de type **short** doit forcément être compris entre les valeurs -32 768 et 32 768.

### *Le type byte*

Le type **byte** est basé sur le type **short** si ce n'est qu'un nombre de type **byte** doit forcément être compris entre les valeurs -128 et 127.

### Le type nonPositiveInteger

Basé sur le type **integer**, le type **nonPositiveInteger** représente un nombre entier qui n'est pas positif. Concrètement, cela correspond à un nombre négatif ou au nombre zéro.

Voyons ensemble un exemple :

**Code : XML**

```
<xsd:element name="nombre" type="xsd:nonPositiveInteger" />
```

**Code : XML**

```
<nombre>-42</nombre>

<nombre>0</nombre>

<nombre>-00042</nombre>
```

### Le type negativeInteger

Basé sur le type **nonPositiveInteger**, le type **negativeInteger** représente un nombre entier strictement négatif, c'est-à-dire strictement inférieur à zéro.

### Le type nonNegativeInteger

Basé sur le type **integer**, le type **nonNegativeInteger** représente un nombre entier qui n'est pas négatif, c'est-à-dire un nombre supérieur ou égal à zéro.

Soit l'exemple suivant :

**Code : XML**

```
<xsd:element name="nombre" type="xsd:nonPositiveInteger" />
```

**Code : XML**

```
<nombre>42</nombre>

<nombre>0</nombre>

<nombre>+00042</nombre>
```

### Le type positiveInteger

Basé sur le type **nonNegativeInteger**, le type **positiveIntegerreprésente** un nombre entier strictement positif, c'est-à-dire strictement supérieur à zéro.

### Le type unsignedLong

Le type **unsignedLong** est basé sur le type **nonNegativeInteger** et représente un entier compris entre les valeurs 0 et 18 446 744 073 709 551 615.

### *Le type unsignedInt*

Le type **unsignedInt** est basé sur le type **unsignedLong** et représente un entier compris entre les valeurs 0 et 4 294 967 295.

### *Le type unsignedShort*

Le type **unsignedShort** est basé sur le type **unsignedInt** et représente un entier compris entre les valeurs 0 et 65 535.

### *Le type unsignedByte*

Le type **unsignedByte** est basé sur le type **unsignedShort** et représente un entier compris entre les valeurs 0 et 255.

## Les autres types

## Le tableau récapitulatif

| Type | Description |
|---|---|
| boolean | représente l'état vrai ou faux |
| QName | représente un nom qualifié |
| NOTATION | représente une notation |
| anyURI | représente une URI |
| base64Binary | représente une donnée binaire au format Base64 |
| hexBinary | représente une donnée binaire au format hexadecimal |

## Plus en détails

### *Le type boolean*

Le type **boolean**, comme son nom le laisse deviner, représente un booléen. Pour ceux qui ignore de quoi il s'agit, un booléen permet d'indiquer l'un des 2 états suivant : vrai ou faux.

Si 2 états sont possibles, 4 valeurs sont en réalités acceptées :

- true qui représente l'état vrai ;
- false qui représente l'état faux ;
- 1 qui représente l'état vrai ;
- 0 qui représente l'état faux ;

Conscient que cette notion n'est pas forcément facile à comprendre lorsque c'est la première fois qu'on la rencontre, je vais tenter de l'illustrer avec un exemple. Notre exemple va nous permettre, via un attribut, de savoir si une personne est un animal ou pas.

**Code : XML**

```
<xsd:attribute name="animal" type="xsd:boolean" />
```

**Code : XML**

```
<!-- Victor Hugo n'est pas un animal -->
```

```xml
<personne animal="false">Victor Hugo</personne>

<!-- Zozor est bien un animal -->
<personne animal="true">Zozor</personne>
```

### Le type QName

Le type **QName** représente un nom qualifié. C'est un concept qui s'appuie sur l'utilisation des espaces de nom.

### Le type NOTATION

Le type **NOTATION** permet d'identifier et décrire du contenu XML ou non comme par exemple une image.

### Le type anyURI

Comme son nom l'indique, le type **anyURI** représente une URI (Uniform Resource Identifier). Une URI est une chaîne de caractère qui permet d'identifier une ressource.

On distingue généralement 2 types d'URI :

- les URL (Uniform Resource Locator) est probablement la forme d'URI la plus connue et je suis sûr que ce nom vous parle. En effet, les URL sont utilisés pour décrire l'adresse d'une ressource sur un réseau. Par exemple http://www.siteduzero.com et ftp://ftp.rfc-editor.org/in-notes/rfc2396.txt sont deux URL possibles.
- les URN (Uniform Resource Name) sont quant à eux utilisés pour identifier une ressource dans un espace de nom. Je ne vais pas m'attarder sur les URN car nous reviendrons plus tard dans ce cours sur la notion des espaces de nom.

Une URI pour identifier une ressource de manière relative ou absolue.

Voyons ensemble quelques exemples :

**Code : XML**

```xml
<xsd:attribute name="adresse" type="xsd:anyURI"/>
```

**Code : XML**

```xml
<!-- URI absolu -->
<image
adresse="http://www.siteduzero.com/bundles/common/images/spreadsheetV32.png"
/>

<!-- URI relatif -->
<image adresse="../bundles/common/images/spreadsheetV32.png"/>
```

### Le type base64Binary

Le type **base64Binary** représente une donnée binaire au format Base64. Vous trouverez plus d'informations à ce sujet sur Wikipedia.

Comme de nombreux types que nous avons vu, le type base64Binary impose le respect de plusieurs règles :

- seuls les lettres (majuscules ou minuscules), les chiffres et les symboles + / et = sont autorisés ;
- le nombre de caractères qui composent la chaîne doit être un multiple de 4.

Dans le cas où le symbole = est utilisé, de nouvelles règles doivent être respectées :

- il ne peut apparaître qu'en fin de chaîne, une fois ou deux ;
- dans le cas où il est utilisé qu'une seule fois, il doit forcément être précédé des caractères A Q g ou w ;
- dans le cas où il est utilisé 2 fois, il doit forcément être précédé des caractères A E I M Q U Y c g k o s w 0 (zéro) 4 ou 8.

### *Le type hexBinary*

Le type **hexBinary** représente une donnée binaire au format hexadecimal.

Si comme pour le type base64Binary quelques règles sont à respecter, elles sont bien plus simples dans le cas du type **hexBinary**. Ainsi, seuls les lettres entre A et F (majuscules ou minuscules), ainsi que les chiffres sont autorisés. A noter que le nombre de caractères composant la chaîne doit forcément être un multiple de 2.

#  Schéma XML : les types complexes

Jusqu'à maintenant, nous avons vu ce qu'est un élément simple. Il est maintenant temps de passer le niveau au dessus et découvrir ensemble ce qu'est un élément complexe !

## Définition

### Bref rappel

Au cours des chapitres précédents, nous avons vu ensemble ce qu'est un élément simple, à savoir un élément qui ne contient qu'une valeur dont le type est dit simple. Un élément simple ne contient pas d'autres éléments ni aucun attribut.

Nous avons également vu comment déclarer un élément simple ainsi qu'un attribut. Cependant nous n'avons pas vu comment déclarer un attribut dans un élément. En effet, un élément qui possède un attribut n'est plus un élément simple. On parle alors d'élément complexe.

### Les éléments complexes

Un élément complexe est un élément qui contient d'autres éléments ou des attributs. Bien évidemment les éléments contenus dans un éléments peuvent également contenir des éléments ou des attributs. J'espère que vous suivez toujours !

Je vous propose de voir quelques exemples d'éléments XML qui dans un Schéma XML sont considérés comme complexes :

**Code : XML**

```
<!-- la balise personne contient d'autres balises => élément
complexe -->
<personne>
    <!-- la balise nom est un élément simple -->
    <nom>ROBERT</nom>
    <!-- la balise prenom est un élément simple -->
    <prenom>Axel</prenom>
</personne>

<!-- la balise personne possède un attribut => élément complexe -->
<personne sexe="feminin">Axel ROBERT</personne>
```

Dans les prochaines lignes, je vous propose de voir différents exemples qui vont nous permettre de voir et comprendre comment déclarer des éléments complexes dans un Schéma XML.

## Déclarer un élément complexe

Si vous souhaitez déclarer une balise en tant qu'élément complexe, c'est le mot clef **complexType** qu'il faut utiliser associé à celui que nous connaissons déjà : **element**. N'oubliez pas de précéder son utilisation par **xsd:**

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:complexType>
        <!-- contenu ici -->
    <xsd:complexType>
</xsd:element>
```

Nous reviendrons juste après sur la notion de contenu, ne vous inquiétez pas.
Reprenons alors l'un des éléments de type complexe que nous avons vu un peu plus haut :

**Code : XML**

```xml
<personne>
    <nom>ROBERT</nom>
    <prenom>Axel</prenom>
</personne>
```

Voici comment le déclarer :

**Code : XML**

```xml
<xsd:element name="personne">
    <xsd:complexType>
        <!-- contenu ici -->
    </xsd:complexType>
</xsd:element>
```

## Les contenus des types complexes

Concernant les types complexes, il est important de noter qu'il existe 3 types de contenus possibles :

- les contenus simples ;
- les contenus "standards" ;
- les contenus mixtes.

## Les contenus simples
## Définition

Le premier type de contenu possible pour un élément complexe est le contenu simple.

On appelle contenu simple, le contenu d'un élément complexe qui n'est composé que d'attributs et d'un texte de type simple.

## Quelques exemples

Je vous propose de voir quelques exemples d'éléments complexes dont le contenu est dit simple.

**Code : XML**

```xml
<!-- contient un attribut et du texte -->
<prix devise="euros">35</prix>

<!-- contient un attribut et du texte -->
<voiture marque="Renault">Clio</voiture>
```

## Du côté du Schéma XML

### *La syntaxe*

Pour déclarer un élément complexe faisant référence à une balise contenant des attributs et du texte, voici la syntaxe à utiliser :

**Code : XML**

```xml
<xsd:element name="mon_nom">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="mon_type">
```

```
                <xsd:attribute name="mon_nom" type="mon_type" />
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

### *Un exemple*

Reprenons l'exemple d'un prix prenant pour attribut une devise :

**Code : XML**

```
<prix devise="euros">35</prix>
```

Voici alors le schéma XML associé :

**Code : XML**

```
<xsd:element name="prix">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="xsd:positiveInteger">
                <xsd:attribute name="devise" type="xsd:string" />
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

Dans le cas où la balise que l'on cherche à décrire contient plusieurs attributs, il convient de tout simplement les lister entre les balises `<xsd:extension/>`. Par exemple :

**Code : XML**

```
<voiture marque="Renault" type="essence">Clio</voiture>
```

**Code : XML**

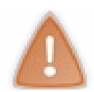
```
<xsd:element name="voiture">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribut name="marque" type="xsd:string" />
                <xsd:attribut name="type" type="xsd:string" />
            </xsd:extension>
        <xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

Comme vous pouvez le constater, on se contente de mettre à la suite les différents attributs qui composent l'élément. À noter que l'ordre dans lequel les attributs sont déclarés dans le Schéma XML n'a aucune importance.

## Les contenus "standards"

## Définition

Après les contenus simples, nous allons monter la barre d'un cran et nous attaquer aux contenus "standards".



Il est important de noter que cette appellation n'est nullement officielle. C'est une appellation maison car il s'agit du cas de figure qui à tendance à revenir le plus souvent.

Ce que j'appelle contenu "standard", c'est le contenu d'un élément complexe qui n'est composé que d'autres éléments (simples ou complexes) ou uniquement d'attributs.

## Quelques exemples

Comme pour le contenu simple, voyons quelques exemples de contenu "standard" :

**Code : XML**

```
<!-- contient d'autres éléments -->
<personne>
    <nom>DUPONT</nom>
    <prenom>Robert</prenom>
</prenom>

<!-- contient un attribut -->
<voiture marque="Renault" />
```

## Balise contenant un ou plusieurs attributs

Je vous propose de débuter par le cas de figure le plus simple, à savoir celui d'un élément complexe qui ne contient que des attributs.

Reprenons l'exemple de notre voiture du dessus :

**Code : XML**

```
<voiture marque="Renault" />
```

Voici alors le Schéma XML associé :

**Code : XML**

```
<xsd:element name="voiture">
    <xsd:complexType>
        <xsd:attribut name="marque" type="xsd:string" />
    </xsd:complexType>
</xsd:element>
```

Il n'y a pour le moment rien de bien compliqué. On se contente d'imbriquer une balise `<xsd:attribut />` dans une balise `<xsd:complexType />`.

Si l'on tente de complexifier un petit peu nous allons nous rendre compte que dans le fond, rien ne change. Prenons par exemple le cas d'une balise contenant plusieurs attributs :

**Code : XML**

```
<voiture marque="Renault" modele="Clio" />
```

Regardons alors le Schéma XML :

**Code : XML**

```
<xsd:element name="voiture">
    <xsd:complexType>
        <xsd:attribut name="marque" type="xsd:string" />
        <xsd:attribut name="modele" type="xsd:string" />
    </xsd:complexType>
</xsd:element>
```

Comme vous pouvez le constater, on se contente de mettre à la suite les différents attributs qui composent l'élément. Une fois de plus, l'ordre dans lequel les balises `<xsd:attribut />` sont placées n'a aucune importance.

## Balise contenant d'autres éléments

Il est maintenant temps de passer à la suite et de jeter un coup d'œil aux balises qui contiennent d'autres éléments.

### *La séquence*

Une **séquence** est utilisée lorsque l'on souhaite spécifier que les éléments contenus dans un type complexe doivent apparaître dans un ordre précis.

Voici comment se déclare une **séquence** au niveau d'un Schéma XML :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:complexType>
        <xsd:sequence>
            <!-- liste des éléments -->
        </xsd:sequence>
        <!-- listes des attributs -->
    </xsd:complexType>
</xsd:element>
```

Voyons tout de suite un exemple :

**Code : XML**

```
<xsd:element name="personne">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="nom" type="xsd:string"/>
            <xsd:element name="prenom" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="sexe" type="xsd:string" />
    </xsd:complexType>
</xsd:element>
```

Cet extrait signifie que la balise `<personne />` qui possède l'attribut **sexe**, contient les balises `<nom />` et `<prenom />`

dans cet ordre.

Illustrons alors cet exemple :

**Code : XML**

```
<!-- valide -->
<personne sexe="masculin">
    <nom>DUPONT</nom>
    <prenom>Robert</prenom>
</personne>

<!-- invalide => les balises nom et prenom sont inversées -->
<personne sexe="masculin">
    <prenom>Robert</prenom>
    <nom>DUPONT</nom>
</personne>
```

### *Le type all*

Le type **all** est utilisé lorsque l'on veut spécifier que les éléments contenu dans un type complexe peuvent apparaître dans n'importe quel ordre. Ils doivent cependant tous apparaître une et une seule fois.

Voici comment se déclare le type **all** au niveau d'un Schéma XML :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:complexType>
        <xsd:all>
            <!-- liste des éléments -->
        </xsd:all>
        <!-- listes des attributs -->
    </xsd:complexType>
</xsd:element>
```

Voyons tout de suite un exemple :

**Code : XML**

```
<xsd:element name="personne">
    <xsd:complexType>
        <xsd:all>
            <xsd:element name="nom" type="xsd:string"/>
            <xsd:element name="prenom" type="xsd:string"/>
        </xsd:all>
    </xsd:complexType>
</xsd:element>
```

Cet extrait signifie donc que la balise `<personne />` contient les balises `<nom />` et `<prenom />` dans n'importe quel ordre.

Illustrons alors cet exemple :

**Code : XML**

```
<!-- valide -->
<personne sexe="masculin">
```

```
    <nom>DUPONT</nom>
    <prenom>Robert</prenom>
</prenom>

<!-- valide -->
<personne sexe="masculin">
    <prenom>Robert</prenom>
    <nom>DUPONT</nom>
</prenom>
```

### *Le choix*

Un **choix** est utilisé lorsque l'on veut spécifier qu'un élément contenu dans un type complexe soit choisi dans une liste pré-définie.

Voici comment se déclare un **choix** au niveau d'un Schéma XML :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:complexType >
        <xsd:choice>
            <!-- liste des éléments -->
        </xsd:choice>
        <!-- listes des attributs -->
    </xsd:complexType>
</xsd:element>
```

Voyons sans plus tarder un exemple :

**Code : XML**

```
<xsd:element name="personne">
    <xsd:complexType>
        <xsd:choice>
            <xsd:element name="nom" type="xsd:string"/>
            <xsd:element name="prenom" type="xsd:string"/>
        </xsd:choice>
    </xsd:complexType>
</xsd:element>
```

Cet extrait signifie donc que la balise `<personne />` contient soit la balise `<nom />`, soit `<prenom />`.

Illustrons cet exemple :

**Code : XML**

```
<!-- valide -->
<personne sexe="masculin">
    <nom>DUPONT</nom>
</prenom>

<!-- valide -->
<personne sexe="masculin">
    <prenom>Robert</prenom>
</prenom>

<!-- invalide => les 2 balises prenom et nom ne peuvent pas
apparaître en même temps -->
```

```
<personne sexe="masculin">
    <prenom>Robert</prenom>
    <nom>DUPONT</nom>
</prenom>
```

## Cas d'un type complexe encapsulant un type complexe

Avant de terminer cette partie, il nous reste un cas à voir : celui d'un type complexe encapsulant également un type complexe.

Prenons par exemple le document XML suivant :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<personne>
    <identite>
        <nom>NORRIS</nom>
        <prenom>Chuck</prenom>
    </identite>
</personne>
```

Ce document XML permet d'identifier une personne via son nom et son prénom. Voyons alors le Schéma XML qui définit notre document XML :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="personne">
        <xsd:complexType>
     <xsd:sequence>
         <xsd:element name="identite">
      <xsd:complexType>
          <xsd:sequence>
       <xsd:element name="nom"  type="xsd:string"/>
       <xsd:element name="prenom"  type="xsd:string"/>
   </xsd:sequence>
      </xsd:complexType>
  </xsd:element>
     </xsd:sequence>
 </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

En soit, il n'y a rien de compliqué. Il convient juste de repérer, que lorsqu'on place un élément complexe au sein d'un autre élément complexe, dans notre cas, une **identité** dans une **personne**, il convient d'utiliser une **séquence**, un **choix** ou un type **all**.

## Les contenus mixtes

## Définition

Il est temps de conclure ce chapitre avec le dernier type de contenu possible : les contenus mixtes.

Un contenu mixte est le contenu d'un élément complexe qui est composé d'attributs, d'éléments et de texte.

## Un exemple

Afin d'illustrer cette définition, je vous propose de nous appuyer sur un exemple :

**Code : XML**

```
<balise1>
  Ceci est une chaîne de caractères
  <balise2>10</balise2>
  7.5
</balise1>
```

## Du côté du Schéma XML

### *La syntaxe*

Pour déclarer un élément complexe au contenu mixte, voici la syntaxe à utiliser :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:complexType  mixed="true">
        <!-- liste des éléments -->
    </xsd:complexType>
    <!-- liste des attributs -->
</xsd:element>
```

La nouveauté est donc l'utilisation du mot clef **mixed**.

### *Un exemple*

Prenons l'exemple d'une facture fictive dans laquelle on souhaite identifier l'acheteur et la somme qu'il doit payer.

**Code : XML**

```
<facture><acheteur>Zozor</acheteur>, doit payer <somme>1000</somme>€
.</facture>
```

Voici comment le traduire au sein d'un Schéma XML :

**Code : XML**

```
<xsd:element name="facture">
    <xsd:complexType name="" mixed="true">
        <xsd:sequence>
            <element name="acheteur" type="xsd:string" />
            <element name="somme" type="xsd:int" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Comme vous pouvez le remarquer, j'ai utilisé la balise `<xsd:sequence />` pour encapsuler la liste des balises contenues dans la balise `<facture />`, mais vous pouvez bien évidemment adapter à votre cas de figure et choisir parmi les balises que nous avons vu dans le chapitre précédent, à savoir :

- `<xsd:sequence />` ;

- **`<xsd:all />`** ;
- **`<xsd:choice />`**.

Après cette découverte des éléments de type complexe, je vous propose de ne pas nous arrêter là et d'approfondir un peu nos connaissances !

#  Schéma XML : aller plus loin

Il en maintenant temps de s'attaquer au dernier chapitre portant sur les Schémas XML.

Ce chapitre va aborder différentes notions qu'il est important de connaître et surtout comprendre pour affiner l'écriture de Schéma XML.

## Le nombre d'occurrences

Dans le chapitre précédent, nous avons vu comment écrire des éléments de type complexe. Je peux maintenant vous avouer que je vous ai caché quelques petites choses.

La première que nous allons voir concerne le nombre d'occurrences d'une balise. Pour vous aider à bien comprendre cette notion, je vous propose d'étudier un morceau de Schéma XML que nous avons déjà vu. Il s'agit de celui d'une personne qui possède un nom et un prénom :

**Code : XML**

```
<xsd:complexType name="personne">
    <xsd:sequence>
        <xsd:element name="nom" type="xsd:string"/>
        <xsd:element name="prenom" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Comme je vous le disais, cet extrait signifie que la balise `<personne />` contient les balises `<nom />` et `<prenom />` dans cet ordre.

La notion d'occurrence va nous permettre de préciser si les balises, dans le cas de notre exemple `<nom />` et `<prenom />`, peuvent apparaître plusieurs fois, voir pas du tout.

### Le cas par défaut

Le cas par défaut est celui que nous avons vu jusqu'à maintenant. Lorsque le nombre d'occurrences n'est pas précisé, la balise doit apparaître une et une seule fois.

### Le nombre minimum d'occurrences

Pour indiquer le nombre minimum d'occurrences d'un élément, on utilise l'attribut `minOccurs`. Comme nous l'avons déjà vu plus haut, sa valeur par défaut est 1. A noter que dans le cas où il est utilisé, sa valeur doit obligatoirement être supérieure à zéro.

### Le nombre maximum d'occurrences

Pour indiquer le nombre maximum d'occurrences d'un élément, on utilise l'attribut `maxOccurs`. Comme pour le nombre minimum d'occurrence, la valeur par défaut est 1. Une nouvelle fois, dans le cas où il est utilisé, sa valeur doit obligatoirement être supérieure à zéro. A noter qu'il également possible de ne pas spécifier un nombre maximal d'occurrence grâce au mot clef `unbounded`.

### Exemple

Je vous propose de terminer ce chapitre en l'illustrant à l'aide d'un exemple.

**Code : XML**

```
<xsd:complexType name="personne">
    <xsd:sequence>
        <xsd:element name="nom" type="xsd:string" />
```

```
        <xsd:element name="prenom" type="xsd:string" minOccurs="2"
maxOccurs="unbounded" />
    </xsd:sequence>
</xsd:complexType>
```

Dans l'extrait de Schéma XML ci-dessus, on remarque que pour l'élément `prenom`, le nombre minimum d'occurrences est à 2 tandis qu'il n'y a pas de maximum. Cela signifie, que dans notre fichier XML, cette balise devra apparaître entre 2 et une infinité de fois comme en témoigne les extraits de fichier XML suivants :

**Code : XML**

```
<personne>
    <nom>Zozor</nom>
    <prenom>Robert</prenom>
    <prenom>Bernard</prenom>
</personne>

<personne>
    <nom>Zozor</nom>
    <prenom>Robert</prenom>
    <prenom>Bernard</prenom>
    <prenom>Paul</prenom>
    <prenom>Pierre</prenom>
</personne>
```

## La réutilisation des élements

Avant d'attaquer la notion d'héritage, nous allons voir comment réutiliser des types complexes afin d'en écrire le moins possible. C'est bien connu, les informaticiens sont des fainéants !

## Pourquoi ne pas tout écrire d'un seul bloc ?

Puisqu'un exemple est souvent bien meilleur que de longues explications, je vous propose d'étudier le document XML suivant :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<banque>
    <!-- 1er client de la banque -->
    <client>
        <!-- identité du client -->
        <identite>
            <nom>NORRIS</nom>
            <prenom>Chuck</prenom>
        </identite>

        <!-- liste des comptes bancaires du client -->
        <comptes>
            <livretA>
                <montant>2500</montant>
            </livretA>
            <courant>
                <montant>4000</montant>
            </courant>
        </comptes>
    </client>
</banque>
```

Ce document XML représente une banque et ses clients. Pour chaque client, on connaît son identité, le montant de son livret A ainsi que le montant de son compte courant.

Avec nos connaissances actuelles, voici ce à quoi ressemble le Schéma XML qui décrit ce document XML :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="banque">
        <xsd:complexType >
            <xsd:sequence>
                <xsd:element name="client">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="identite"
maxOccurs="unbounded" >
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <xsd:element name="nom"
type="xsd:string" />
                                        <xsd:element name="prenom"
type="xsd:string" />
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                            <xsd:element name="comptes">
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <xsd:element name="livretA">
                                            <xsd:complexType>
                                                <xsd:sequence>
                                                    <xsd:element
name="montant" type="xsd:double" />
                                                </xsd:sequence>
                                            </xsd:complexType>
                                        </xsd:element>
                                        <xsd:element name="courant">
                                            <xsd:complexType>
                                                <xsd:sequence>
                                                    <xsd:element
name="montant" type="xsd:double" />
                                                </xsd:sequence>
                                            </xsd:complexType>
                                        </xsd:element>
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Cette construction d'un seul bloc, également appelé construction "en poupées russes" n'est pas des plus lisible. Afin de rendre notre Schéma XML un peu plus lisible et compréhensible, je vous propose de le diviser. Sa lecture n'en sera que facilité !

## Diviser un Schéma XML

### *Quelques explications*

Dans cette partie nous allons voir comment "casser" cette écriture "en poupées russes" afin de rendre notre Schéma XML plus compréhensible, plus accessible.

L'idée est dans l'ensemble assez simple, on déclare de manière globale les différents qui composent notre Schéma XML, puis dans nos structures complexes, on y fait référence.

### *La syntaxe*

La déclaration d'un élément ne change pas par rapport à ce que nous avons vu jusqu'à maintenant, qu'il s'agisse d'un élément simple ou d'un élément complexe.

Établir une référence est alors très simple grâce au mot clef **ref** :

**Code : XML**

```
<xsd:element ref="mon_nom" />
```

Je vous propose d'illustrer l'utilisation de ce nouveau mot clef via un exemple. Nous allons décomposer l'identité d'un client. Pour rappel, voici ce que nous avons écrit dans notre premier essai :

**Code : XML**

```
<xsd:element name="identite" maxOccurs="unbounded" >
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="nom" type="xsd:string" />
            <xsd:element name="prenom" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Nous allons donc faire une déclaration globale des éléments **nom** et **prenom** afin d'y faire référence dans l'élément complexe **identite**.

**Code : XML**

```
<!-- déclaration globale de certains éléments -->
<xsd:element name="nom" type="xsd:string" />
<xsd:element name="prenom" type="xsd:string" />

<xsd:element name="identite" maxOccurs="unbounded" >
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="nom" />
            <xsd:element ref="prenom" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Comme vous pouvez le constater, c'est déjà plus lisible. Cette méthode nous permet également de réutiliser des éléments qui reviennent plusieurs fois comme par exemple l'élément **montant** qui revient plusieurs fois à savoir dans le compte courant et le livret A d'un client.

### *Mise à jour de notre banque et ses clients*

Mettons alors à jour notre Schéma XML afin de le découper le plus possible :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!-- déclaration des éléments -->
    <xsd:element name="nom" type="xsd:string" />
    <xsd:element name="prenom" type="xsd:string" />
    <xsd:element name="montant" type="xsd:double" />

    <xsd:element name="banque">
        <xsd:complexType >
            <xsd:sequence>
                <xsd:element name="client">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="identite"
maxOccurs="unbounded" >
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <xsd:element ref="nom" />
                                        <xsd:element ref="prenom" />
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                            <xsd:element name="comptes">
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <xsd:element name="livretA">
                                            <xsd:complexType>
                                                <xsd:sequence>
                                                    <xsd:element
ref="montant" />
                                                </xsd:sequence>
                                            </xsd:complexType>
                                        </xsd:element>
                                        <xsd:element name="courant">
                                            <xsd:complexType>
                                                <xsd:sequence>
                                                    <xsd:element
ref="montant" />
                                                </xsd:sequence>
                                            </xsd:complexType>
                                        </xsd:element>
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Comme vous pouvez le constater, c'est mieux, mais en ce qui me concerne, je trouve que ce n'est pas encore ça. Plutôt que de déclarer globalement uniquement des éléments simples comme le **montant**, le **nom** ou le **prenom**, pourquoi ne pas déclarer globalement des éléments complexes comme l'identité du client ou encore le livret A ou le compte courant.

Voyons alors le résultat :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!-- déclaration des éléments -->
    <xsd:element name="montant" type="xsd:double" />

    <xsd:element name="identite" maxOccurs="unbounded" >
```

```
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="nom" type="xsd:string" />
                    <xsd:element name="prenom" type="xsd:string" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="livretA">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="montant" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="courant">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="montant" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="comptes">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="livretA" />
                    <xsd:element ref="courant" />
                </xsd:sequence>
            </xsd:complexType>
         </xsd:element>

        <xsd:element name="client">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="identite" />
                    <xsd:element ref="comptes" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <!-- Schéma XML -->
        <xsd:element name="banque">
            <xsd:complexType >
                <xsd:sequence>
                    <xsd:element ref="client" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
```

C'est tout de même plus lisible ! 

## Créer ses propres types

Grâce aux références, nous sommes arrivés à un résultat satisfaisant, mais si l'on regarde vraiment en détail, on se rend vite compte que notre Schéma XML n'est pas optimisé. En effet, les différents comptes de notre client, à savoir le livret A et le compte courant, ont des structures identiques et pourtant, nous les déclarons 2 fois.

Dans cette partie, nous allons donc apprendre à créer nos propres types pour encore et toujours en écrire le moins possible !

### *La syntaxe*

Déclarer un nouveau type, n'est pas plus compliqué que ce que nous avons vu jusqu'à présent. Il est cependant important de noter une petite chose. Les types que nous allons créer peuvent être de 2 natures : simple ou complexe.

Débutons avec la création et l'utilisation d'un type simple. La création d'un type simple est utile lorsque par exemple dans un Schéma XML, plusieurs chaînes de caractères ou plusieurs nombres ont des restrictions similaires.

**Code : XML**

```
<!-- création -->
<xsd:simpleType name="mon_type_perso">
    <xsd:restriction base="mon_type">
        <!-- liste des restrictions -->
    </xsd:restriction>
</xsd:simpleType>

<!-- utilisation-->
<xsd:element name="mon_nom" type="mon_type_perso" />
```

Continuons avec la création et l'utilisation d'un type complexe :

**Code : XML**

```
<!-- création -->
<xsd:ComplexeType name="mon_type_perso">
    <!-- personnalisation du type complexe -->
</xsd:ComplexeType>

<!-- utilisation-->
<xsd:element name="mon_nom" type="mon_type_perso" />
```

### *Mise à jour de notre banque et ses clients*

Je vous propose de mettre à jour notre Schéma XML en créant un type "**compte**" que l'on pourra utiliser pour le **livret A** et le **compte courant** des clients de notre banque.

Voici alors ce que ça donne :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!-- déclaration des éléments -->
    <xsd:element name="montant" type="xsd:double" />

    <xsd:element name="identite" >
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="nom" type="xsd:string" />
                <xsd:element name="prenom" type="xsd:string" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="compte">
        <xsd:sequence>
            <xsd:element ref="montant" />
        </xsd:sequence>
    </xsd:complexType>
```

```
        <xsd:element name="comptes">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="livretA" type="compte" />
                    <xsd:element name="courant" type="compte" />
                </xsd:sequence>
            </xsd:complexType>
         </xsd:element>

        <xsd:element name="client">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="identite" />
                    <xsd:element ref="comptes" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <!-- Schéma XML -->
        <xsd:element name="banque">
            <xsd:complexType >
                <xsd:sequence>
                    <xsd:element ref="client" maxOccurs="unbounded" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
```

Notre Schéma XML est bien plus lisible maintenant ! 

## L'héritage

L'héritage est un concept que l'on retrouve dans la plupart des langages de programmation orienté objet. Certes le XML n'est pas un langage de programmation, mais ça ne l'empêche pas d'embarquer cette notion.

Pour faire simple, l'héritage permet de réutiliser des éléments d'un Schéma XML pour en construire de nouveaux. Si vous avez encore du mal à comprendre le concept, ne vous inquiétez pas, nous allons utiliser des exemples afin d'illustrer ce nouveau concept.

En XML, 2 types d'héritages sont possibles :

- par restriction ;
- par restriction ;

Dans les 2 cas, c'est le mot clef **base** que nous utiliserons pour indiquer un héritage.

## L'héritage par restriction

La première forme d'héritage que nous allons voir est celle par restriction.

### *Définition*

Une restriction est une notion qui peut s'appliquer aussi bien aux éléments qu'aux attributs et qui permet de déterminer plus précisément la valeur attendue via la détermination d'un certain nombre de contraintes.

Par exemple, jusqu'à maintenant, nous sommes capables de dire qu'un élément ou qu'un attribut doit contenir un nombre entier strictement positif. Grâce aux restrictions, nous allons pouvoir pousser le concept jusqu'au bout en indiquant qu'un élément qu'un attribut doit contenir un nombre entier strictement positif compris entre 1 et 100.

Lorsque l'on déclare une restriction sur un élément, la syntaxe suivante doit être respectée :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <!-- détail de la restriction -->
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

La syntaxe est quasiment identique dans le cas d'un attribut :

**Code : XML**

```
<xsd:attribute name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <!-- détail de la restriction -->
        </xsd:restriction>
    </xsd:simpleType>
</xsd:attribute>
```

Vouas avez très probablement remarqués l'attribut `base` dans la balise `<restriction />`. Il s'agit du type de votre balise et donc de votre restriction. Il suffit donc de choisir un type simple dans la liste de ceux que nous avons vu dans les chapitres précédents (`xsd:string`, `xsd:int`, etc.).

### *Le tableau récapitulatif*

| Nom de la restriction | Description |
|---|---|
| minExclusive | permet de définir une valeur minimale exclusive |
| minInclusive | permet de définir une valeur minimale inclusive |
| maxExclusive | permet de définir une valeur maximale exclusive |
| maxInclusive | permet de définir une valeur maximale inclusive |
| totalDigits | permet de définir le nombre exact de chiffres qui composent un nombre |
| fractionDigits | permet de définir le nombre de chiffres autorisés après la virgule |
| length | permet de définir le nombre exact de caractères d'une chaîne |
| minLength | permet de définir le nombre minimum de caractères d'une chaîne |
| maxLength | permet de définir le nombre maximum de caractères d'une chaîne |
| enumeration | permet d'énumérer la liste des valeurs possibles |
| whiteSpace | permet de déterminer le comportement à adopter avec les espaces |
| pattern | permet de définir des expressions rationnelles |

### *Plus en détails*



Pour décrire la syntaxe des différentes restrictions, je vais m'appuyer sur la balise `<xsd:element />`, mais n'oubliez pas que toutes les règles que nous voyons dans ce chapitre sont également valables pour la balise `<xsd:attribute />`.

### *La restriction minExclusive*

La restriction **minExclusive** s'applique à un élément de type numérique et permet de définir sa valeur minimale. Comme son nom le laisse deviner, la valeur indiquée est exclue des valeurs que peut prendre l'élément.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:minExclusive value="ma_valeur" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

Afin d'illustrer cette première restriction, prenons par exemple l'âge d'une personne que l'on souhaite obligatoirement majeure :

**Code : XML**

```
<xsd:complexType name="personne">
    <xsd:attribute name="age">
        <xsd:simpleType>
            <xsd:restriction base="xsd:nonNegativeInteger">
                <xsd:minExclusive value="17" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
</xsd:complexType>
```

**Code : XML**

```
<!-- valide -->
<personne age="18" />

<!-- valide -->
<personne age="43" />

<!-- invalide -->
<personne age="17" />
```

### *La restriction minInclusive*

La restriction **minInclusive** ressemble énormément à la restriction **minExclusive** que nous venons de voir si ce n'est que la valeur indiquée peut-être prise par l'élément.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:minInclusive value="ma_valeur" />
```

```
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

### La restriction maxExclusive

La restriction **maxExclusive** s'applique à un élément de type numérique et permet de définir sa valeur maximale. Comme son nom le laisse deviner, la valeur indiquée est exclue des valeurs que peut prendre l'élément.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:maxExclusive value="ma_valeur" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

### La restriction maxInclusive

La restriction **maxInclusive** ressemble énormément à la restriction **maxExclusive** que nous venons de voir si ce n'est que la valeur indiquée peut-être prise par l'élément.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:maxInclusive value="ma_valeur" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

### La restriction totalDigits

La restriction **totalDigits** s'applique à un élément de type numérique et permet de définir le nombre exact de chiffres qui composent le nombre. Sa valeur doit obligatoirement être supérieure à zéro.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:totalDigits value="ma_valeur" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

Je vous propose d'illustrer tout de suite cette restriction avec un exemple. Une nouvelle fois prenons l'âge d'une personne. Imaginons un contexte dans lequel l'âge d'une personne doit obligatoirement être compris entre 10 et 99. Il doit donc être obligatoirement composé de 2 chiffres :

**Code : XML**

```
<xsd:complexType name="personne">
    <xsd:attribute name="age">
        <xsd:simpleType>
            <xsd:restriction base="xsd:nonNegativeInteger">
                <xsd:totalDigits value="2" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
</xsd:complexType>
```

**Code : XML**

```
<!-- valide -->
<personne age="18" />

<!-- valide -->
<personne age="43" />

<!-- invalide -->
<personne age="4" />
```

## *La restriction fractionDigits*

La restriction **fractionDigits** s'applique à un élément de type numérique et permet de définir le nombre maximal de chiffres qui composent une décimale. Sa valeur peut-être supérieure ou égale à zéro.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:fractionDigits value="ma_valeur" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

## *La restriction length*

La restriction **length** permet de définir le nombre exact de caractères qui composent une chaîne. La valeur renseignée doit obligatoirement être supérieure ou égale à zéro.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
```

```
                <xsd:length value="ma_valeur" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>
```

Pour illustrer l'utilisation de la La restriction **length**, prenons par exemple une empreinte SHA1. Pour faire simple, une empreinte SHA1 est un nombre hexadécimal composé de 40 caractères :

**Code : XML**

```
    <xsd:complexType name="sha1">
        <xsd:simpleType>
            <xsd:restriction base="xsd:hexBinary">
                <xsd:length value="40" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:complexType>
```

**Code : XML**

```
    <!-- valide -->
    <sha1>edf7a6029d6bdfb68447677a1d76639725f795f1</sha1>

    <!-- valide -->
    <sha1>a94a8fe5ccb19ba61c4c0873d391e987982fbbd3</sha1>

    <!-- invalide -->
    <sha1>test</sha1>
```

### *La restriction minLength*

La restriction **minLength** permet de définir le nombre minimum de caractères qui composent une chaîne. La valeur renseignée doit obligatoirement être supérieure ou égale à zéro.

Voici sa syntaxe :

**Code : XML**

```
    <xsd:element name="mon_nom">
        <xsd:simpleType>
            <xsd:restriction base="type_de_base">
                <xsd:minLength value="ma_valeur" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>
```

### *La restriction maxLength*

La restriction **maxLength** permet de définir le nombre maximum de caractères qui composent une chaîne. La valeur renseignée doit obligatoirement être supérieure ou égale à zéro.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:maxLength value="ma_valeur" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

### La restriction enumeration

La restriction **enumeration**, comme son nom le laisse deviner, cette restriction permet d'énumérer la liste des valeurs possibles pour un élément ou un attribut.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:enumeration value="valeur1" />
            <xsd:enumeration value="valeur2" />
            <!-- liste des valeurs... -->
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

Afin d'illustrer cette restriction, prenons l'exemple d'une personne comme nous l'avons déjà fait à plusieurs reprises dans ce tutoriel. Une personne possède un nom, un prénom, et est normalement un homme ou une femme. Écrivons alors la règle d'un Schéma XML permettant de décrire une personne :

**Code : XML**

```
<xsd:complexType name="personne">
    <xsd:sequence>
        <xsd:element name="nom" type="xsd:string"/>
        <xsd:element name="prenom" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="sexe">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="masculin" />
                <xsd:enumeration value="feminin" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
</xsd:complexType>
```

**Code : XML**

```
<!-- valide -->
<personne sexe="masculin">
    <nom>DUPONT</nom>
    <prenom>Robert</prenom>
</personne>

<!-- valide -->
<personne sexe="feminin">
```

```
        <nom>DUPONT</nom>
        <prenom>Robert</prenom>
    </personne>

    <!-- invalide -->
    <personne sexe="pomme">
        <nom>DUPONT</nom>
        <prenom>Robert</prenom>
    </personne>
```

## *La restriction whiteSpace*

La restriction **whiteSpace** permet de spécifier le comportement à adopter par un élément lorsqu'il contient des espaces. Les espaces peuvent être de différentes formes comme par exemple les tabulations, les retours à la ligne, etc.

3 valeurs sont possibles :

- **preserve** : cette valeur permet de préserver tous les espaces ;
- **replace** : cette valeur permet de remplacer tous les espaces (tabulations, retour à la ligne, etc.) par des espaces "simples" ;
- **collapse** : cette valeur permet de supprimer les espaces en début et fin de chaîne, de remplacer les tabulations et les retours à la ligne par un espace "simple" et de remplacer les espaces multiples par un espace "simple".

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:whiteSpace value="ma_valeur" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

## *La restriction pattern*

La restriction **pattern** permet de définir des expressions rationnelles (également appelées expressions régulières). Une expression rationnelle est un motif qui permet de décrire le contenu attendu. Ainsi à l'aide des expressions rationnelles, il est possible de dire que le contenu attendu doit forcément débuter par une majuscule, se terminer un point, débuter par un caractère spécial, ne pas contenir la lettre "z", etc.

Les expressions rationnelles sont un langage à part entière et nous n'allons pas revenir en détail sur ce langage. Mais ne vous inquiétez pas, vous trouverez facilement sur Internet de nombreuses ressources francophones sur le sujet.

Voici sa syntaxe :

**Code : XML**

```
<xsd:element name="mon_nom">
    <xsd:simpleType>
        <xsd:restriction base="type_de_base">
            <xsd:pattern value="ma_valeur" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

Afin d'illustrer cette dernière restriction, prenons l'exemple du prénom non composé d'une personne. On souhaite qu'il débute par une majuscule et soit suivi par plusieurs caractères minuscules :

**Code : XML**

```
<xsd:complexType name="prenom">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="[A-Z][a-z]+" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:complexType>
```

**Code : XML**

```
<!-- valide -->
<prenom>Robert</prenom>

<!-- valide -->
<prenom>Zozor</prenom>

<!-- invalide -->
<prenom>bernard</prenom>
```

## L'héritage par extension

Dans cette seconde partie, nous allons voir le second type d'héritage : l'héritage par extension.

### *Définition*

Une extension est une notion qui permet d'ajouter des informations à un type existant. On peut par exemple vouloir ajouter un élément ou un attribut.

Lorsque l'on déclare une extension sur un élément, c'est toujours le mot clef "base" qui est utilisé :

**Code : XML**

```
<!-- contenu complexe -->
<xsd:complexType name="mon_nom">
    <xsd:complexContent>
        <xsd:extension base="type_de_base">
            <!-- détail de l'extension -->
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<!-- contenu simple -->
<xsd:complexType name="mon_nom">
    <xsd:simpleContent>
        <xsd:extension base="type_de_base">
            <!-- détail de l'extension -->
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

### *Exemple*

Je vous propose de mettre en application cette nouvelle notion d'héritage par extension au travers d'un exemple. Reprenons les données clientes d'une banque que nous manipulions dans le chapitre précédent. Dans cette exemple, nous avions défini un type "**compte**" appliqué au **compte courant** et au **livret A** de notre client :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!-- déclaration des éléments -->
    <xsd:element name="montant" type="xsd:double" />

    <xsd:element name="identite" maxOccurs="unbounded" >
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="nom" type="xsd:string" />
                <xsd:element name="prenom" type="xsd:string" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="compte">
        <xsd:sequence>
            <xsd:element ref="montant" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="comptes">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="livretA" type="compte" />
                <xsd:element name="courant" type="compte" />
            </xsd:sequence>
        </xsd:complexType>
     </xsd:element>

    <xsd:element name="client">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="identite" />
                <xsd:element ref="comptes" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <!-- Schéma XML -->
    <xsd:element name="banque">
        <xsd:complexType >
            <xsd:sequence>
                <xsd:element ref="client" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Imaginons maintenant que le compte courant et le livret A soient un peu différents. Par exemple, un compte courant n'a généralement pas de taux d'intérêts tandis que le livret A en a un. Malgré ce petit changement, le livret A et le compte courant restent sensiblement identiques. C'est là que l'héritage par extension intervient. Nous allons étendre le type compte en y ajoutant un attribut pour créer ainsi un nouveau type : celui d'un compte avec des intérêt.

**Code : XML**

```
<!-- le montant -->
<xsd:element name="montant" type="xsd:double" />
```

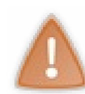
```
<!-- compte sans intérêts -->
<xsd:complexType name="compte">
    <xsd:sequence>
        <xsd:element ref="montant" />
    </xsd:sequence>
</xsd:complexType>

<!-- compte avec intérêts grâce à l'héritage par extension -->
<xsd:complexType name="compteInteret">
    <xsd:complexContent>
        <xsd:extension base="compte">
            <xsd:attribute name="interet" type="xsd:float" />
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

## Les identifiants

Nous avons déjà vu que dans un Schéma XML, il est possible d'identifier des ressources et d'y faire référence grâce aux mots clefs **ID** et **IDREF**.

Il est cependant possible d'aller plus loin et d'être encore plus précis grâce à 2 nouveaux mots clefs : **key** et **keyref**.



Pour bien comprendre la suite de chapitre, il est nécessaire de connaître le fonctionnement d'une technologie qu'on appelle **XPath**, technologie que nous aborderons en plus en détail dans la prochaine partie.
Pour le moment, retenez simplement que cette technologie nous permet de sélectionner avec précision des éléments formant un document XML.

## La syntaxe

### *L'élément key*

Au sein d'un Schéma XML, l'élément `<key />` est composé :

- d'un élément `<selector />` contenant une expression XPath afin d'indiquer l'élément à référencer ;
- d'un ou plusieurs éléments `<field />` contenant une expression XPath afin d'indiquer l'attribut servant d'identifiant ;

Ce qui nous donne :

**Code : XML**

```
<xsd:key name="nom_identifiant">
    <selector xpath="expression_XPath" />
    <!-- liste d'éléments field -->
    <field xpath="expression_XPath" />
</xsd:key>
```

### *L'élément keyref*

L'élément `<keyref />` se construit sensiblement comme l'élément `<key />`. Il est donc composé :

- d'un élément `<selector />` contenant une expression XPath afin d'indiquer l'élément à référencer ;
- d'un ou plusieurs éléments `<field />` contenant une expression XPath afin d'indiquer l'attribut servant d'identifiant ;

Ce qui nous donne :

**Code : XML**

```
<xsd:keyref name="nom" refer="nom_identifiant">
    <selector xpath="expression_XPath" />
    <!-- liste d'éléments field -->
    <field xpath="expression_XPath" />
</xsd:keyref>
```

## Exemple

Afin d'illustrer cette nouvelle notion, je vous propose de d'étudier le document XML suivant :

**Code : XML**

```
<famille>
    <pere id="PER-1" />
    <enfant id="PER-2" pere="PER-1" />
</famille>
```

Dans cet exemple, une famille est composée d'un père et d'un enfant dont chacun possède un identifiant unique au travers de l'attribut `id`. L'enfant possède également un attribut **pere** qui contient l'identifiant de son père.

Je vous propose de construire ensemble le Schéma XML correspond au document XML. Commençons par décrire l'élément `<pere />` :

**Code : XML**

```
<xsd:element name="pere">
    <xsd:complexType>
        <xsd:attribut name="id" type="xsd:NCName" />
    </xsd:complexType>
</xsd:element>
```

Continuons avec l'élément `<enfant />` :

**Code : XML**

```
<xsd:element name="enfant">
    <xsd:complexType>
        <xsd:attribut name="id" type="xsd:NCName" />
        <xsd:attribut name="pere" type="xsd:NCName" />
    </xsd:complexType>
</xsd:element>
```

Terminons avec l'élément `<famille />` :

**Code : XML**

```
<xsd:element name="famille">
    <xsd:complexType>
```

```
            <xsd:sequence>
                <xsd:element ref="pere" />
                <xsd:element ref="enfant" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
```

Modifions ce dernier élément afin d'y ajouter nos identifiants. Pour le moment, je vous demande d'accepter les expressions XPath présentes dans le Schéma XML. Dans la partie suivante, vous serez normalement en mesure de les comprendre.

**Code : XML**

```
    <xsd:element name="famille">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="pere" />
                <xsd:element ref="enfant" />
            </xsd:sequence>
        </xsd:complexType>

        <!-- identifiant du père -->
        <xsd:key name="pereId">
            <xsd:selector xpath="./pere" />
            <xsd:field xpath="@id" />
        </xsd:key>

        <!-- identifiant de l'enfant -->
        <xsd:key name="enfantId">
            <xsd:selector xpath="./enfant" />
            <xsd:field xpath="@id" />
        </xsd:key>

        <!-- référence à l'identifiant du père dans l'élément enfant --
    >
        <xsd:key name="pereIdRef" refer="pereId">
            <xsd:selector xpath="./enfant" />
            <xsd:field xpath="@pere" />
        </xsd:key>
    </xsd:element>
```

## Un exemple avec EditiX

Afin de terminer ce chapitre, je vous propose de voir ensemble comment écrire un Schéma XML avec EditiX.

Pour faire simple, je vous propose de reprendre l'exemple de notre banque vu dans le chapitre sur la réutilisation des types.

## Création du document XML

La création du document XML n'a rien de bien compliqué puisque nous l'avons déjà vu ensemble dans la partie précédente.

Pour ceux qui ne s'en souviennent pas, vous pouvez y jeter un coup d'œil ici.

Voici le document que vous devez écrire :

**Code : XML**

```
    <?xml version="1.0" encoding="UTF-8"?>
    <banque xsi:noNamespaceSchemaLocation="banque.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <!-- 1er client de la banque -->
        <client>
            <!-- identité du client -->
```

```
            <identite>
                <nom>NORRIS</nom>
                <prenom>Chuck</prenom>
            </identite>

            <!-- liste des comptes bancaires du client -->
            <comptes>
                <livretA>
                    <montant>2500</montant>
                </livretA>
                <courant>
                    <montant>4000</montant>
                </courant>
            </comptes>
        </client>
    </banque>
```

Si vous essayez de lancer la vérification du document, vous devriez normalement avoir le message d'erreur suivant (voir la figure suivante).

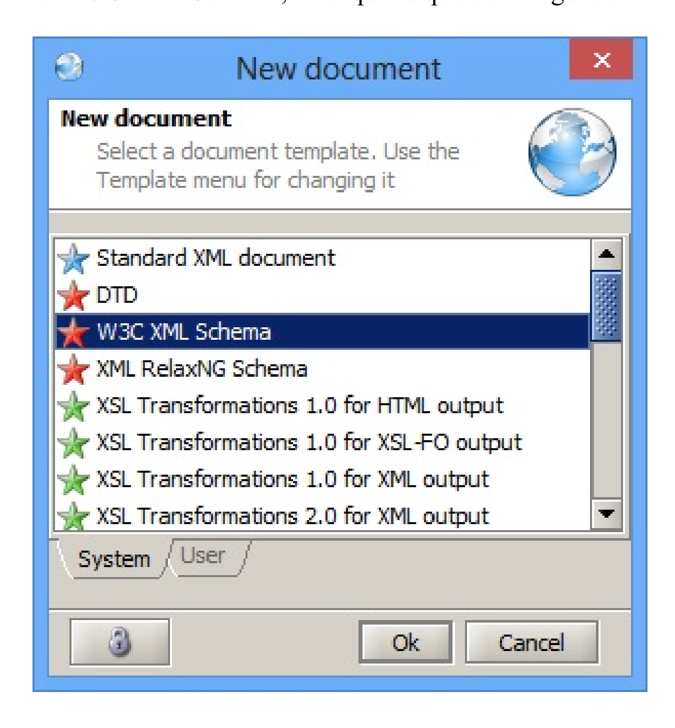



Ce message est pour le moment complètement normal puisque nous n'avons pas encore créé notre document XSD.

## Création du document XSD

Pour créer un nouveau document, vous pouvez sélectionner dans la barre de menu **File** puis **New** ou encore utiliser le raccourci clavier Ctrl + N.

Dans la liste qui s'affiche, sélectionnez **W3C XML Schema**, ainsi qu'indiqué sur la figure suivante.




Votre document XSD n'est normalement pas vierge. Voici ce que vous devriez avoir :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">


</xs:schema>
```

Replacez le contenu par notre véritable Schéma XML :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!-- déclaration des éléments -->
    <xsd:element name="montant" type="xsd:double" />

    <xsd:element name="identite" >
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="nom" type="xsd:string" />
                <xsd:element name="prenom" type="xsd:string" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:complexType name="compte">
        <xsd:sequence>
            <xsd:element ref="montant" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="comptes">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="livretA" type="compte" />
                <xsd:element name="courant" type="compte" />
            </xsd:sequence>
        </xsd:complexType>
     </xsd:element>

    <xsd:element name="client">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="identite" />
                <xsd:element ref="comptes" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <!-- Schéma XML -->
    <xsd:element name="banque">
        <xsd:complexType >
            <xsd:sequence>
                <xsd:element ref="client" maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

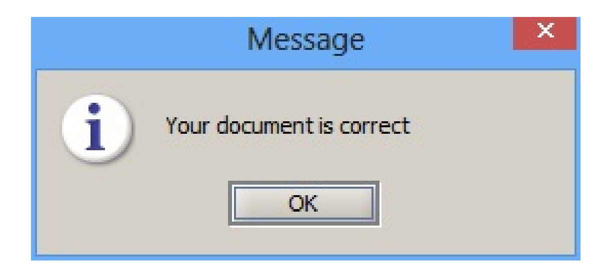
Enregistrez ensuite votre document avec le nom **banque.xsd** au même endroit que votre document XML.

## Vérification du Schéma XML

Vous pouvez vérifier que votre Schéma XML n'a pas d'erreur de syntaxe en sélectionnant dans la barre de menu **DTD/Schema**

puis **Check this DTD** ou encore en utilisant le raccourci clavier Ctrl + K.

Vous devriez normalement avoir le message suivant (voir la figure qui suit).

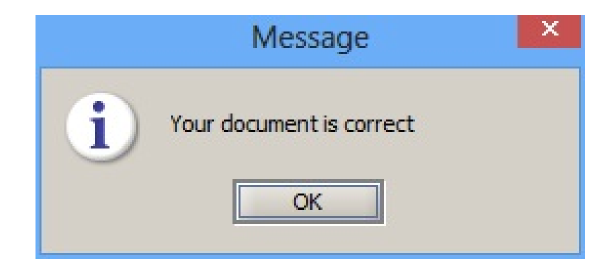



## Vérification du document XML

Il est maintenant temps de vérifier que le document XML est **valide** !

Pour ce faire, sélectionnez dans la barre de menu **XML** puis **Check this document** ou utilisez le raccourci clavier Ctrl + K.

Le message suivant doit normalement s'afficher (voir la figure suivante).



#  TP : Schéma XML d'un répertoire

Comme pour chaque technologie que nous voyons ensemble au cours de ce tutoriel, je vous propose de conclure avec un TP !

Ce TP est dans la même lignée que celui sur les définitions DTD puisqu'il s'agit d'écrire le Schéma XML d'un répertoire téléphonique.

## L'énoncé

Le but de ce TP est de créer le Schéma XML du répertoire que nous avons déjà vu.

Pour rappel, voici les informations que l'on souhaite connaître pour chaque personne :

- son sexe (homme ou femme) ;
- son nom ;
- son prénom ;
- son adresse ;
- un ou plusieurs numéros de téléphone (téléphone portable, fixe, bureau, etc.) ;
- une ou plusieurs adresses e-mail (adresse personnelle, professionnelle, etc.).

Voici le document XML que nous avions construit :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <adresse>
            <numero>7</numero>
            <voie type="impasse">impasse du chemin</voie>
            <codePostal>75015</codePostal>
            <ville>PARIS</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
        <emails>
            <email type="personnel">john.doe@wanadoo.fr</email>
            <email type="professionnel">john.doe@societe.com</email>
        </emails>
    </personne>

    <!-- Marie POPPINS -->
    <personne sexe="feminin">
        <nom>POPPINS</nom>
        <prenom>Marie</prenom>
        <adresse>
            <numero>28</numero>
            <voie type="avenue">avenue de la république</voie>
            <codePostal>13005</codePostal>
            <ville>MARSEILLE</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="bureau">04 05 06 07 08</telephone>
        </telephones>
        <emails>
            <email type="professionnel">contact@poppins.fr</email>
        </emails>
```

```
        </personne>
    </repertoire>
```

## Une solution

Comme à chaque fois, je vous fais part de ma solution !

Le fichier XML avec le Schéma XML référencé :

**Code : XML**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<repertoire xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="repertoire.xsd">
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <adresse>
            <numero>7</numero>
            <voie type="impasse">impasse du chemin</voie>
            <codePostal>75015</codePostal>
            <ville>PARIS</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
        <emails>
            <email type="personnel">john.doe@wanadoo.fr</email>
            <email type="professionnel">john.doe@societe.com</email>
        </emails>
    </personne>

    <!-- Marie POPPINS -->
    <personne sexe="feminin">
        <nom>POPPINS</nom>
        <prenom>Marie</prenom>
        <adresse>
            <numero>28</numero>
            <voie type="avenue">avenue de la république</voie>
            <codePostal>13005</codePostal>
            <ville>MARSEILLE</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="bureau">04 05 06 07 08</telephone>
        </telephones>
        <emails>
            <email type="professionnel">contact@poppins.fr</email>
        </emails>
    </personne>
</repertoire>
```

Le fichier XSD :

**Code : XML**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!-- balises isolées -->
    <xsd:element name="nom" type="xsd:string"/>
    <xsd:element name="prenom" type="xsd:string"/>
```

```
<!-- balises d'une adresse -->
<xsd:element name="numero" type="xsd:string"/>
<xsd:element name="voie">
    <xsd:complexType>
 <xsd:simpleContent>
     <xsd:extension base="xsd:string">
  <xsd:attribute name="type">
      <xsd:simpleType>
   <xsd:restriction base="xsd:string">
       <xsd:enumeration value="impasse"/>
<xsd:enumeration value="avenue"/>
<xsd:enumeration value="rue"/>
                 </xsd:restriction>
 </xsd:simpleType>
  </xsd:attribute>
 </xsd:extension>
 </xsd:simpleContent>
</xsd:complexType>
</xsd:element>
<xsd:element name="codePostal">
    <xsd:simpleType>
 <xsd:restriction base="xsd:string">
     <xsd:pattern value="[0-9]{5}"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
<xsd:element name="ville" type="xsd:string"/>
<xsd:element name="pays" type="xsd:string"/>

<!-- balise adresse -->
<xsd:element name="adresse">
    <xsd:complexType>
 <xsd:sequence>
     <xsd:element ref="numero"/>
     <xsd:element ref="voie"/>
     <xsd:element ref="codePostal"/>
     <xsd:element ref="ville"/>
     <xsd:element ref="pays"/>
 </xsd:sequence>
</xsd:complexType>
</xsd:element>

<!-- balise telephone -->
<xsd:element name="telephone">
<xsd:complexType>
        <xsd:simpleContent>
     <xsd:extension base="xsd:string">
  <xsd:attribute name="type">
      <xsd:simpleType>
   <xsd:restriction base="xsd:string">
       <xsd:enumeration value="fixe"/>
<xsd:enumeration value="portable"/>
<xsd:enumeration value="bureau"/>
   </xsd:restriction>
 </xsd:simpleType>
  </xsd:attribute>
 </xsd:extension>
        </xsd:simpleContent>
</xsd:complexType>
</xsd:element>

<!-- balise telephones -->
<xsd:element name="telephones">
    <xsd:complexType>
 <xsd:sequence>
     <xsd:element ref="telephone" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

```
        <!-- balise email -->
        <xsd:element name="email">
            <xsd:complexType>
         <xsd:simpleContent>
             <xsd:extension base="xsd:string">
          <xsd:attribute name="type">
              <xsd:simpleType>
           <xsd:restriction base="xsd:string">
               <xsd:enumeration value="personnel"/>
        <xsd:enumeration value="professionnel"/>
           </xsd:restriction>
       </xsd:simpleType>
          </xsd:attribute>
      </xsd:extension>
         </xsd:simpleContent>
     </xsd:complexType>
        </xsd:element>

        <!-- balise emails -->
        <xsd:element name="emails">
     <xsd:complexType>
                <xsd:sequence>
             <xsd:element ref="email" maxOccurs="unbounded"/>
         </xsd:sequence>
     </xsd:complexType>
        </xsd:element>

        <!--balise personne -->
        <xsd:element name="personne">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="nom"/>
                    <xsd:element ref="prenom"/>
                    <xsd:element ref="adresse"/>
                    <xsd:element ref="telephones"/>
                    <xsd:element ref="emails"/>
                </xsd:sequence>

                <!-- attribut sexe -->
                <xsd:attribute name="sexe">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                            <xsd:enumeration value="masculin"/>
                            <xsd:enumeration value="feminin"/>
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:attribute>
            </xsd:complexType>
        </xsd:element>

        <!-- Schéma XML -->
        <xsd:element name="repertoire">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element ref="personne" maxOccurs="unbounded" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
```

## Un bref commentaire

Dans cette solution, je suis allé au plus simple. Libre à vous de créer de nouvelles règles si vous souhaitez par exemple utiliser un

pattern précis pour les numéros de téléphone ou les adresses e-mails.

# Partie 3 : Traitez vos données XML



# DOM : Introduction à l'API

Au cours des deux premières parties de ce tutoriel, nous avons vu comment écrire un document XML et sa définition. Cette troisième partie sera quant à elle consacrée à leur exploitation logicielle.

## Qu'est-ce que L'API DOM ?

## La petite histoire de DOM

**DOM** ou **Document Object Model** de son nom complet est ce qu'on appelle un **parseur XML**, c'est-à-dire une technologie grâce à laquelle il est possible de lire un document XML et d'en extraire différentes informations (éléments, attributs, commentaires, etc...) afin de les exploiter.

Comme pour la plupart des technologies abordées dans ce tutoriel, DOM est un standard du W3C et ce depuis sa première version en 1998. A moment où j'écris ces lignes, la technologies en est à sa troisième version.

Il est très important de noter que DOM est une recommandation complètement indépendante de toute plate-forme et langage de programmation. Au travers de DOM, le W3C fournit une recommandation, c'est-à-dire une manière d'exploiter les documents XML.

Aujourd'hui, la plupart des langages de programmation propose leur implémentation de DOM :

- C ;
- C++ ;
- Java ;
- C# ;
- Perl ;
- PHP ;
- etc.



Dans les chapitres suivants, les exemples seront illustrés à l'aide du langage Java.

## L'arbre XML

Dans le chapitre précédent, je vous disais que DOM est une technologie complètement indépendante de toute plate-forme et langage de programmation et qu'elle se contente de fournir une manière d'exploiter les documents XML.

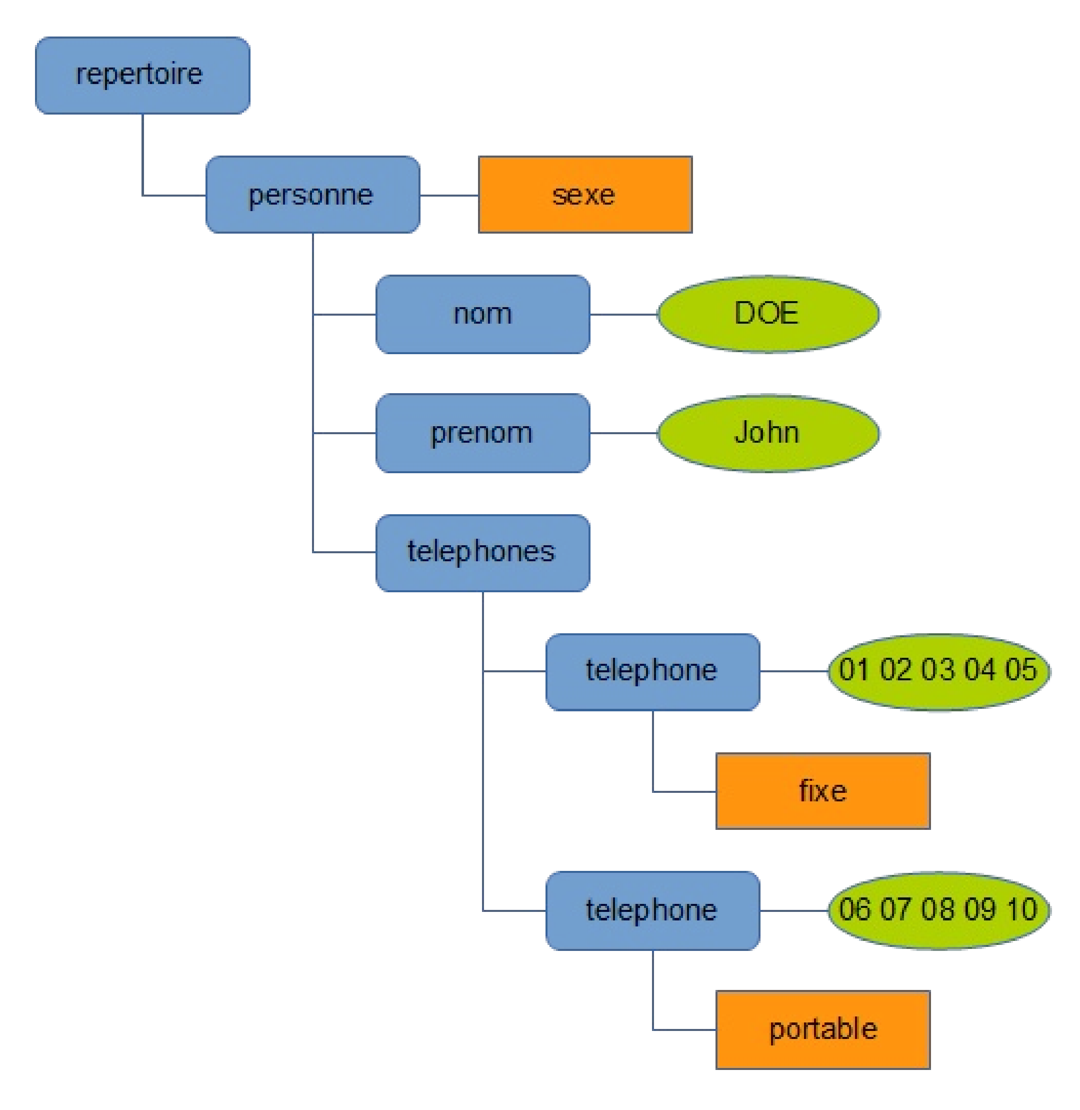
En réalité, lorsque votre document XML est lu par un parseur DOM, le document est représenté en mémoire sous la forme d'un arbre dans lequel les différents éléments sont liés les uns aux autres par une relation parent/enfant. Il est ensuite possible de passer d'un élément à un autre via un certain nombre de fonctions que nous verrons dans le chapitre suivant.

Je vous propose d'illustrer cette notion d'arbre grâce à un exemple. Soit le document XML suivant :

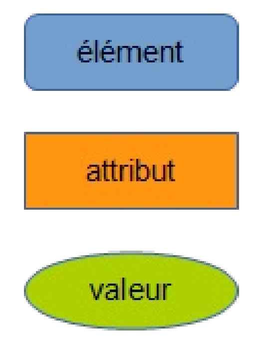
**Code : XML**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
    </personne>
</repertoire>
```

Voici à la figure suivante ce à quoi ressemble l'arbre une fois modélisé.



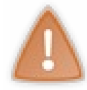

Pour bien comprendre à quoi correspondent les couleurs et les formes, voici la légende en figure suivante.




Dans nos futurs programmes c'est donc ce genre d'arbres que nous allons parcourir afin d'obtenir les informations que l'on souhaite exploiter en passant d'un élément à un autre. Mais avant de voir comment procéder, je vous propose de revenir sur le vocabulaire utilisé par DOM.

## Le vocabulaire et les principaux éléments

Dans ce chapitre, nous allons découvrir ensemble le vocabulaire utilisé par le parseur DOM. Nous allons également en profiter pour faire le tour des principaux éléments en terme de programmation.



Lors de la description des différents éléments, je vais tenter d'être le plus neutre possible en me détachant de tout langage de programmation. Cependant, des connaissances en programmation objet sont nécessaires pour comprendre ce chapitre.

## Document

### *Définition*

Le **document** comme son nom le laisse deviner désigne le document XML dans son ensemble. Il est donc composé :

- du prologue ;
- du corps.

### *La classe*

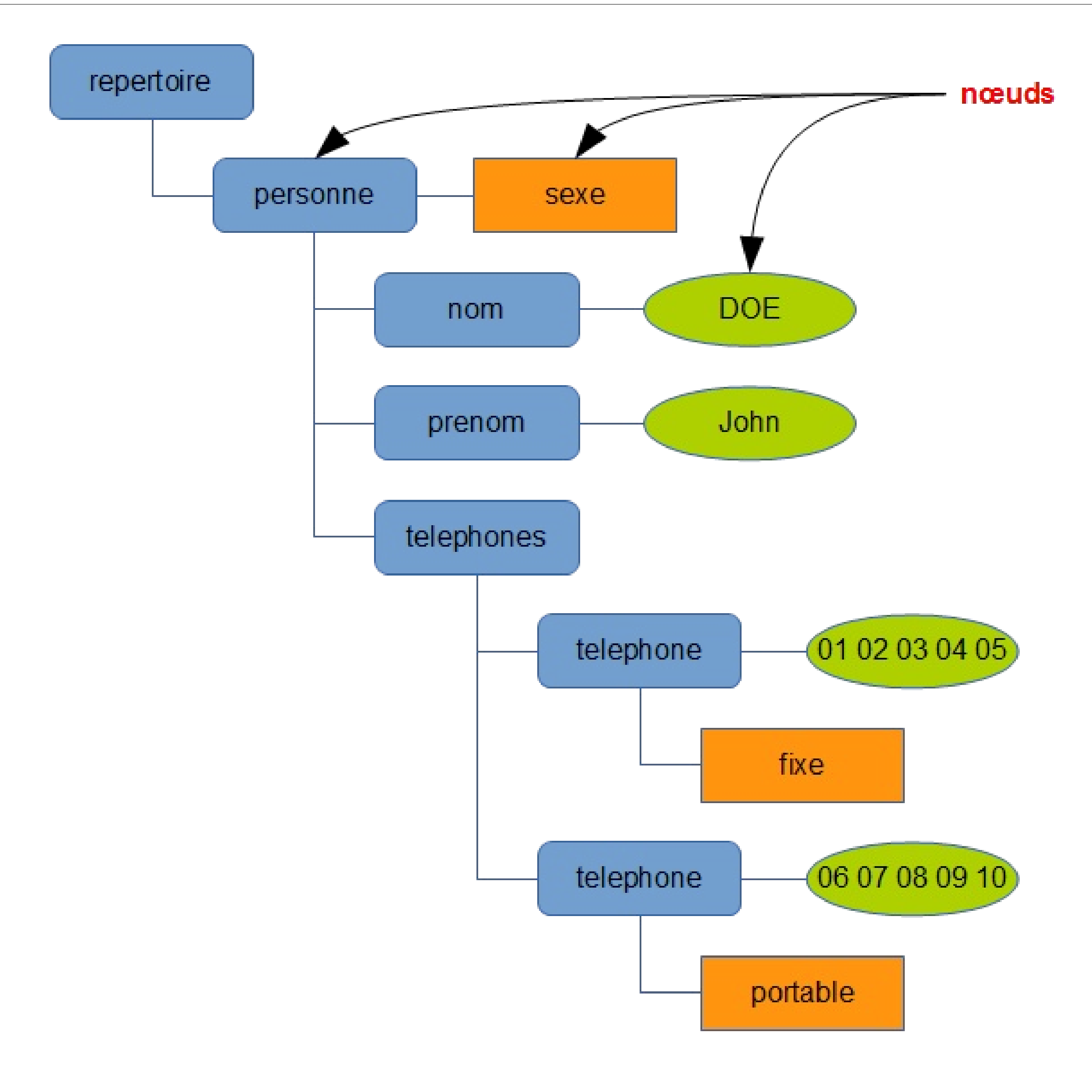
Grâce à la classe **Document** nous allons pouvoir exploiter aussi bien le prologue que le corps de nos documents XML. Cette classe va également se révéler indispensable lorsque nous allons vouloir créer ou modifier des documents XML. En effet, via les nombreuses méthodes proposées, nous allons pouvoir ajouter des éléments, des commentaires, des attributs, etc.

## Node

### *Définition*

Un **Node** ou **Nœud** en français peut-être véritablement considéré comme l'élément de base d'un arbre XML. Ainsi, toute branche ou feuille est un nœud. Un élément est donc un nœud, tout comme une valeur et un attribut.

Je vous propose de reprendre le schéma vu précédemment et de tenter d'identifier 3 nœuds parmi tous ceux présents (voir la figure suivante).




### *La classe*

La classe **Node** nous permet d'obtenir un certain nombre d'informations lors de l'exploitation d'un document XML. Ainsi, il est possible d'obtenir le type du nœud (attribut, valeur, etc.) son nom, sa valeur, la liste des nœuds fils, le nœud parent, etc. Cette classe propose également un certain nombre de méthodes qui vont nous aider à créer et modifier un document XML en offrant par exemple la possibilité de supprimer un nœud, ou d'en remplacer un par un autre, etc.

## Element

### *Définition*

Un **Element** représente une balise d'un document XML. Si l'on reprend le schéma de l'arbre XML du dessus, les éléments sont en bleu.

### *La classe*

La classe **Element**, en plus de nous fournir le nom de la balise, nous offre de nombreuses fonctionnalités comme par exemple la possibilité de récupérer les informations d'un attribut ou encore de récupérer la listes des noeuds d'un élément portant un nom spécifique.

## Attr

#### *Définition*

Un **Attr** désigne un attribut. Si l'on reprend le schéma de l'arbre XML du dessus, les attributs sont en orange.

#### *La classe*

La classe **Attr**, permet d'obtenir un certain nombre d'information concernant les attributs comme son nom ou encore sa valeur. Cette classe va également nous être utile lorsque l'on voudra créer ou modifier des documents XML.

### Text

#### *Définition*

Un **Text** désigne le contenu d'une balise. Si l'on reprend le schéma de l'arbre XML du dessus, ils sont en vert.

#### *La classe*

En plus de la donnée textuelle, la classe **Text**, permet de facilement modifier un document XML en proposant par exemple des méthodes de suppression ou de remplacement de contenu.

### Les autres éléments

Il est presque impossible de présenter tous les éléments du DOM vu la densité de la technologie. Sachez cependant que nous avons vu les principaux et qu'un exemple d'utilisation de l'implémentation Java est prévu dans le chapitre suivant.

Concernant les éléments non décrits, il existe par exemple la classe **Comment** permettant de gérer les commentaires ou encore la classe **CDATASection** permettant de d'exploiter les sections **CDATA** d'un document XML.

Finalement, sachez que grâce à DOM il est également possible de vérifier la validité d'un document XML à une définition DTD ou un Schéma XML.

#  DOM : Exemple d'utilisation en Java

Dans ce chapitre, nous allons voir pas à pas comment utiliser l'implémentation Java de DOM pour parcourir et créer un fichier XML.

Exceptionnellement, sachez qu'il n'y aura pas de TP sur cette technologie. En effet, DOM étant implémenté dans de nombreux langages de programmation, il est impossible pour moi de proposer une correction dans chacun des langages utilisés !

## Lire un document XML

## Le document XML

Avant de plonger dans le code, voici le document XML que nous allons tenter d'exploiter :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
    </personne>
</repertoire>
```

## Mise en place du code

### *Etape 1 : récupération d'une instance de la classe "DocumentBuilderFactory"*

Avant même de pouvoir prétendre créer un parseur DOM, nous devons récupérer une instance de la classe DocumentBuilderFactory. C'est à partir de cette instance que dans l'étape suivante nous pourrons créer notre parseur.

Pour récupérer une instance de la classe "DocumentBuilderFactory" une ligne de code suffit :

**Code : Java**

```
final DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
```

Cette ligne de code s'accompagne de l'importation du package :

**Code : Java**

```
import javax.xml.parsers.DocumentBuilderFactory;
```

### *Etape 2 : création d'un parseur*

La seconde étape consiste à créer un parseur à partir de notre variable `factory` créée dans l'étape 1.

Une nouvelle fois, une seule ligne de code est suffisante :

**Code : Java**

```java
final DocumentBuilder builder = factory.newDocumentBuilder();
```

Pour pouvoir utiliser la classe DocumentBuilder, le package suivant est nécessaire :

**Code : Java**

```java
import javax.xml.parsers.DocumentBuilder;
```

Puisque l'appel à la fonction newDocumentBuilder(), peut lever un exception, il convient de le placer dans un bloc de type **try/catch** :

**Code : Java**

```java
try {
    final DocumentBuilder builder = factory.newDocumentBuilder();
}
catch (final ParserConfigurationException e) {
    e.printStackTrace();
}
```

La gestion de cette exception oblige à également importer le package :

**Code : Java**

```java
import javax.xml.parsers.ParserConfigurationException;
```

### *Etape 3 : création d'un Document*

La 3ème étape consiste à créer un **Document** à partir parseur de l'étape 2. Plusieurs possibilités s'offre alors à vous :

- à partir d'un fichier ;
- à partir d'un flux.

Ce flux peut par exemple être le résultat de l'appel à un web service. Dans notre exemple, c'est un fichier qui est utilisé :

**Code : Java**

```java
final Document document= builder.parse(new File("repertoire.xml"));
```

Comme à chaque nouvelle instruction, des packages doivent être importés :

**Code : Java**

```java
import java.io.File;
import org.w3c.dom.Document;
```

De nouvelles exceptions pouvant être levées, il convient également de modifier de les capturer dans des blocs **catch**. Voici alors

ce que vous devriez avoir :

**Code : Java**

```
try {
    final DocumentBuilder builder = factory.newDocumentBuilder();
    final Document document= builder.parse(new
File("repertoire.xml"));
}
catch (final ParserConfigurationException e) {
    e.printStackTrace();
}
catch (final SAXException e) {
    e.printStackTrace();
}
catch (final IOException e) {
    e.printStackTrace();
}
```

La gestion de ces nouvelles exceptions nous oblige également à importer quelques packages :

**Code : Java**

```
import java.io.IOException;
import org.xml.sax.SAXException;
```

Comme je vous le disais dans le chapitre précédent, un Document représente le document XML dans son intégralité. Il contient son prologue et son corps. Nous allons pouvoir le vérifier en affichant les éléments du prologues, à savoir :

- la version XML utilisé ;
- l'encodage utilisé ;
- s'il s'agit d'un document "standalone" ou non.

**Code : Java**

```
//Affiche la version de XML
System.out.println(document.getXmlVersion());

//Affiche l'encodage
System.out.println(document.getXmlEncoding());

//Affiche s'il s'agit d'un document standalone
System.out.println(document.getXmlStandalone());
```

A l'exécution du programme, voici ce que vous devriez avoir à l'écran :

**Code : Console**

```
UTF-8
1.0
true
```

Si l'on compare cet affichage au prologue du document XML, on se rend compte que les informations correspondent bien.

### Etape 4 : récupération de l'Element racine

Dans cette 4ème étape, nous allons laisser de côté le prologue et tenté de nous attaquer au corps du document XML. Pour se faire, nous allons extraire l'Element racine du Document. C'est à partir de lui que nous pourrons ensuite naviguer dans le reste du document.

Pour récupérer l'élément racine il suffit d'écrire de faire appel à la fonction getDocumentElement() de notre document :

**Code : Java**

```
final Element racine = document.getDocumentElement();
```

Une nouvelle fois, pour pouvoir utiliser la classe Element, le package suivant doit être importé :

**Code : Java**

```
import org.w3c.dom.Element;
```

Nous pouvons dès maintenant vérifier que ce que nous venons de récupérer est bien l'élément racine de notre document en affichant son nom :

**Code : Java**

```
System.out.println(racine.getNodeName());
```

Après exécution du programme, voici ce que vous devriez avoir à l'écran :

**Code : Console**

```
repertoire
```

### Etape 5 : récupération des personnes

C'est à partir de cette étape que les choses sérieuses commencent ! En effet, dans cette 5ème étape, nous allons réellement parcourir le corps de notre document XML.

Il est possible de parcourir un document XML sans connaître sa structure. Il est donc possible de créer un code assez générique notamment grâce à la récursivité. Cependant, ce n'est pas la méthode que j'ai choisi d'utiliser dans ce chapitre ;). Nous allons donc parcourir notre document partant du principe que nous connaissons sa structure.

Pour récupérer tous les noeuds enfants de la racine, voici la ligne de code à écrire :

**Code : Java**

```
final NodeList racineNoeuds = racine.getChildNodes();
```

Il vous faudra également importer le package suivant :

**Code : Java**

```
import org.w3c.dom.NodeList;
```

Nous pouvons également nous amuser à afficher le nom de chacun des nœuds via le code suivant :

**Code : Java**

```
final int nbRacineNoeuds = racineNoeuds.getLength();

for (int i = 0; i<nbRacineNoeuds; i++) {
    System.out.println(racineNoeuds.item(i).getNodeName());
}
```

Vous devriez alors avoir le résultat suivant :

**Code : Console**

```
#text
#comment
#text
personne
#text
```

On retrouve bien notre balise `<personne />` au milieu d'autres nœuds de type **text** et **comment**. Nous allons maintenant légèrement modifier notre boucle afin de n'afficher à l'écran que les nœuds étant des éléments. Grâce à la méthode `getNodeType()` de la classe **Node** :

**Code : Java**

```
for (int i = 0; i<nbRacineNoeuds; i++) {
    if(racineNoeuds.item(i).getNodeType() == Node.ELEMENT_NODE) {
        final Node personne = racineNoeuds.item(i);
        System.out.println(personne.getNodeName());
    }
}
```

A l'exécution de votre programme, vous devriez normalement avoir le résultat suivant :

**Code : Console**

```
personne
```

A noter que si vous avions voulu récupérer le commentaire, nous aurions comparé le type de notre nœud à la constante `Node.COMMENT_NODE`.

Avant de passer à l'affichage de la suite du document XML, nous allons tenter d'afficher le sexe de la personne, qui pour rappel est un attribut. Il existe plusieurs manières de faire plus ou moins génériques. La méthode la plus générique est de faire appelle à la méthode `getAttributes()` de la classe **Node** qui nous renvoie l'ensemble des attributs du nœud. Dans notre cas, nous allons utiliser la méthode `getAttribute(nom)` qui nous renvoie la valeur de l'attribut spécifié en paramètre. Cette méthode n'est cependant pas accessible à partir de la classe **Node**, il convient donc de "caster" notre nœud en un **Element** pour pouvoir l'appeler :

**Code : Java**

```
        for (int i = 0; i<nbRacineNoeuds; i++) {
            if(racineNoeuds.item(i).getNodeType() == Node.ELEMENT_NODE) {
                final Element personne = (Element) racineNoeuds.item(i);
         System.out.println(personne.getNodeName());
         System.out.println("sexe : " + personne.getAttribute("sexe"));
            }
        }
```

A l'écran devrait alors s'afficher le résultat suivant :

**Code : Console**

```
personne
sexe : masculin
```

### *Etape 6 : récupération du nom et du prénom*

Nous allons maintenant récupérer le nom et le prénom des personnes présentes dans notre document XML. Puisque nous savons exactement ce que l'on souhaite récupérer, nous allons y accéder directement via la méthode `getElementsByTagName(name)` de l'objet **Element**. Cette méthode nous renvoie tous éléments contenus dans l'élément et portant le nom spécifié.

Pour mieux comprendre, voyons un exemple :

**Code : Java**

```
final NodeList noms = personne.getElementsByTagName("nom");
```

Ainsi, nous venons de récupérer tous les éléments d'une personne ayant pour nom "nom". Dans notre cas, nous savons qu'une personne ne peut avoir qu'un seul prénom, nous pouvons donc préciser que nous voulons le premier élément de la liste :

**Code : Java**

```
final Element nom = (Element)
personne.getElementsByTagName("nom").item(0);
```

Finalement, si l'on souhaite afficher le **Text** de la balise, il nous suffit d'appeler la méthode `getTextContent()` :

**Code : Java**

```
System.out.println(nom.getTextContent());
```

Vous devriez alors voir s'afficher à l'écran le nom de la seule personne déclarée dans notre document XML :

**Code : Console**

```
DOE
```

Pour extraire le prénom d'une personne, la logique est exactement la même. 

### *Etape 7 : récupération des numéros de téléphone*

La 7ème et dernière étape de la lecture de notre document XML consiste à récupérer les numéros de téléphone d'une personne. La logique est sensiblement la même que dans l'étape 1 si ce n'est que le résultat de la méthode `getElementsByTagName(name)` nous renverra éventuellement plusieurs résultats. Il suffit alors de boucler sur les résultat pour afficher les valeurs et les attributs.

Voici le code qui devrait être écrit :

**Code : Java**

```
final NodeList telephones =
personne.getElementsByTagName("telephone");
final int nbTelephonesElements = telephones.getLength();

for(int j = 0; j<nbTelephonesElements; j++) {
    final Element telephone = (Element) telephones.item(j);
    System.out.println(telephone.getAttribute("type") + " : " +
telephone.getTextContent());
}
```

Vous devriez alors voir s'afficher à l'écran la liste des numéros de téléphones :

**Code : Console**

```
fixe : 01 02 03 04 05
portable : 06 07 08 09 10
```

## Le code complet

Nous venons donc de lire ensemble notre premier document XML ! Pour ceux qui en auraient besoin, vous trouverez le code complet du petit programme que nous venons d'écrire juste en dessous :

**Code : Java**

```
import java.io.File;
import java.io.IOException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class ReadXMLFile {
    public static void main(final String[] args) {
        /*
* Etape 1 : récupération d'une instance de la classe
"DocumentBuilderFactory"
*/
        final DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
```

```
            try {
                /*
        * Etape 2 : création d'un parseur
        */
                final DocumentBuilder builder =
        factory.newDocumentBuilder();

             /*
        * Etape 3 : création d'un Document
        */
             final Document document= builder.parse(new
        File("repertoire.xml"));

             //Affiche du prologue
             System.out.println("*************PROLOGUE************");
             System.out.println("version : " + document.getXmlVersion());
             System.out.println("encodage : " + document.getXmlEncoding());
                    System.out.println("standalone : " +
        document.getXmlStandalone());

             /*
        * Etape 4 : récupération de l'Element racine
        */
             final Element racine = document.getDocumentElement();

             //Affichage de l'élément racine
             System.out.println("\n*************RACINE************");
             System.out.println(racine.getNodeName());

             /*
        * Etape 5 : récupération des personnes
        */
             final NodeList racineNoeuds = racine.getChildNodes();
             final int nbRacineNoeuds = racineNoeuds.getLength();

             for (int i = 0; i<nbRacineNoeuds; i++) {
                 if(racineNoeuds.item(i).getNodeType() == Node.ELEMENT_NODE)
        {
                     final Element personne = (Element)
        racineNoeuds.item(i);

              //Affichage d'une personne
              System.out.println("\n*************PERSONNE************");
              System.out.println("sexe : " + personne.getAttribute("sexe"));

                  /*
        * Etape 6 : récupération du nom et du prénom
        */
              final Element nom = (Element)
        personne.getElementsByTagName("nom").item(0);
              final Element prenom = (Element)
        personne.getElementsByTagName("prenom").item(0);

              //Affichage du nom et du prénom
              System.out.println("nom : " + nom.getTextContent());
              System.out.println("prénom : " + prenom.getTextContent());

              /*
        * Etape 7 : récupération des numéros de téléphone
        */
              final NodeList telephones =
        personne.getElementsByTagName("telephone");
              final int nbTelephonesElements = telephones.getLength();

              for(int j = 0; j<nbTelephonesElements; j++) {
                  final Element telephone = (Element) telephones.item(j);

                                   //Affichage du téléphone

        System.out.println(telephone.getAttribute("type") + " : " +
```

```
telephone.getTextContent());
            }
              }
          }
              }
              catch (final ParserConfigurationException e) {
                  e.printStackTrace();
              }
              catch (final SAXException e) {
                  e.printStackTrace();
              }
              catch (final IOException e) {
                  e.printStackTrace();
              }
          }
      }
```

## Ecrire un document XML

## Le document XML

Dans le chapitre précédent, nous avons vu comment lire un document XML. Dans ce chapitre, je vous propose d'en créer un de toute pièce. Voici le document que nous allons créer :

**Code : XML**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
    </personne>
</repertoire>
```

Je suis sûr que vous le connaissez. 

## Mise en place du code

### *Etape 1 : récupération d'une instance de la classe "DocumentBuilderFactory"*

Comme pour la lecture d'un document XML, la première étape consiste à récupérer une instance de la classe DocumentBuilderFactory. C'est à partir de cette instance que notre parseur sera créé dans l'étape suivante :

**Code : Java**

```java
final DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
```

N'oubliez pas d'importer le package :

**Code : Java**

```java
import javax.xml.parsers.DocumentBuilderFactory;
```

### *Etape 2 : création d'un parseur*

La seconde étape est également commune à la lecture et la création d'un document XML. Ainsi, nous allons créer un parseur à partir de notre variable `factory` créée dans l'étape précédente :

**Code : Java**

```
final DocumentBuilder builder = factory.newDocumentBuilder();
```

Cette ligne de code s'accompagne de l'importation du package suivant :

**Code : Java**

```
import javax.xml.parsers.DocumentBuilder;
```

Bien que nous l'ayons déjà vu dans le chapitre précédent, n'oubliez pas qu'une exception peut-être levée, c'est pourquoi cette instruction doit être placée dans un bloc de type **try/catch** :

**Code : Java**

```
try {
    final DocumentBuilder builder = factory.newDocumentBuilder();
}
catch (final ParserConfigurationException e) {
    e.printStackTrace();
}
```

La gestion de cette exception nous oblige à également importer le package suivant :

**Code : Java**

```
import javax.xml.parsers.ParserConfigurationException;
```

### *Etape 3 : création d'un Document*

La 3ème étape consiste à créer un **Document** vierge. Ce document est créé à partir de notre parseur :

**Code : Java**

```
final Document document= builder.newDocument();
```

Pour pouvoir utiliser la classe Document, n'oubliez pas d'importer le package suivant :

**Code : Java**

```
import org.w3c.dom.Document;
```

### *Etape 4 : création de l'Element racine*

Dans cette 4ème étape, nous allons créer l'élément racine de notre document XML, à savoir la balise `<repertoire />`. La création de l'élément racine se fait via notre parseur et plus particulièrement la fonction `createElement()` qui prend en paramètre le nom que l'on souhaite donner à la balise :

**Code : Java**

```
final Element racine = document.createElement("repertoire");
```

A noter que l'utilisation de la classe Element s'accompagne de l'importation du package :

**Code : Java**

```
import org.w3c.dom.Element;
```

Maintenant que notre élément racine est déclaré, nous pouvons l'ajouter à notre document :

**Code : Java**

```
document.appendChild(racine);
```

### *Etape 5 : création d'une personne*

Si l'on regarde le document XML que l'on doit créer, on s'aperçoit qu'avant de créer la balise `<personne/>`, nous devons créer un commentaire.

La création d'un commentaire n'est pas plus compliquée que la création d'une balise et se fait via la fonction createComment() du parseur qui prend comme paramètre le fameux commentaire :

**Code : Java**

```
final Comment commentaire = document.createComment("John DOE");
```

Pour pouvoir déclarer un commentaire, n'oubliez pas d'importer le package suivant :

**Code : Java**

```
import org.w3c.dom.Comment;
```

Il convient ensuite d'ajouter notre commentaire à la suite de notre document et plus spécifiquement à la suite de notre élément racine. Si l'on se réfère à l'arbre XML, le commentaire est réellement un sous élément de l'élément racine. C'est pourquoi celui-ci est ajouté en tant qu'enfant de l'élément racine :

**Code : Java**

```
racine.appendChild(commentaire);
```

Nous pouvons maintenant nous attaquer à la création de la balise **<personne />**. Il s'agit d'un élément au même titre que l'élément racine que nous avons déjà vu. Puisque la balise est au même niveau que le commentaire, il convient de l'ajouter en tant qu'enfant de l'élément racine :

**Code : Java**

```
final Element personne = document.createElement("personne");
racine.appendChild(personne);
```

Si l'on s'arrête ici, on omet d'ajouter l'attribut sexe pourtant présent dans la balise **<personne />** du document XML que l'on souhaite créer. Ajouter un attribut à un élément est en réalité très simple et se fait via la méthode setAttribute() de la classe Element. Cette méthode prend 2 paramètres : le nom de l'attribut et sa valeur :

**Code : Java**

```
personne.setAttribute("sexe", "masculin");
```

### *Etape 6 : création du nom et du prénom*

En soit, la création des balises **<nom />** et **<prenom />** n'a rien de compliqué. En effet, nous avons déjà créé ensemble plusieurs éléments.

**Code : Java**

```
final Element nom = document.createElement("nom");
final Element prenom = document.createElement("prenom");

personne.appendChild(nom);
personne.appendChild(prenom);
```

La nouveauté est ici en ce qui concerne le renseignement de la valeur contenu dans les balises, à savoir John DOE dans notre exemple. Pour se faire, il convient d'ajouter à nos balises un enfant de type Text. Cet enfant doit être créé avec la méthode createTextNode() du document qui prend en paramètre la valeur :

**Code : Java**

```
nom.appendChild(document.createTextNode("DOE"));
prenom.appendChild(document.createTextNode("John"));
```

### *Etape 7 : création des numéros de téléphone*

Je vais aller très vite sur cette étape en vous fournissant directement le code source. En effet, cette 7ème étape ne contient rien de nouveau par rapport à ce que nous avons vu jusqu'ici :

**Code : Java**

```
final Element telephones = document.createElement("telephones");

final Element fixe = document.createElement("telephone");
fixe.appendChild(document.createTextNode("01 02 03 04 05"));
fixe.setAttribute("type", "fixe");

final Element portable = document.createElement("telephone");
```

```
        portable.appendChild(document.createTextNode("06 07 08 09 10"));
        portable.setAttribute("type", "portable");

        telephones.appendChild(fixe);
        telephones.appendChild(portable);
        personne.appendChild(telephones);
```

### *Etape 8 : affichage du résultat*

Il est maintenant temps de passer à la dernière étape qui consiste à afficher notre document XML fraîchement créé. 2 possibilités s'offrent à nous :

- dans un document XML ;
- dans la console de l'IDE.

Ne vous inquiétez pas, les 2 possibilités seront abordées dans ce tutoriel. 

Pour pouvoir afficher notre document XML, nous allons avoir besoin de plusieurs objets Java. Le premièr est une instance de la classe TransformerFactory :

**Code : Java**

```java
final TransformerFactory transformerFactory =
TransformerFactory.newInstance();
```

La récupération de cette instance s'accompagne de l'importation du package suivant :

**Code : Java**

```java
import javax.xml.transform.TransformerFactory;
```

Nous allons utiliser cette instance pour créer un objet Transformer. C'est grâce à lui que nous pourrons afficher notre document XML par la suite :

**Code : Java**

```java
final Transformer transformer = transformerFactory.newTransformer();
```

A noter que la fonction newTransformer() peut lever une exception de type TransformerConfigurationException qu'il est importer de capturer dans via un bloc catch.

N'oubliez pas d'importer les packages suivants :

**Code : Java**

```java
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
```

Pour afficher le document XML, nous utiliserons la méthode transform() de notre transformer. Cette méthode prend 2 paramètres :

- la source ;
- la sortie.

**Code : Java**

```
transformer.transform(source, sortie);
```

A noter qu'une exception de type TransformerConfiguration est susceptible d'être levée :

**Code : Java**

```
import javax.xml.transform.TransformerException;
```

En ce qui nous concerne, la source que l'on souhaite afficher est notre document XML. Cependant, nous ne pouvons pas passer notre objet document tel quel. Il convient de le transformer légèrement sous la forme d'un objet DOMSource :

**Code : Java**

```
final DOMSource source = new DOMSource(document);
```

Pour pouvoir utiliser cette classe, il convient d'importer le package suivant :

**Code : Java**

```
import javax.xml.transform.dom.DOMSource;
```

Maintenant que nous avons la source, occupons nous de la sortie. La sortie est en réalité un objet StreamResult. C'est ici que nous allons préciser si nous souhaitons afficher notre document dans un fichier ou dans la console de notre IDE :

**Code : Java**

```
//Code à utiliser pour afficher dans un fichier
final StreamResult sortie = new StreamResult(new
File("F:\\file.xml"));

//Code à utiliser pour afficher dans la console
final StreamResult sortie = new StreamResult(System.out);
```

Encore une fois, l'importation d'un package est nécessaire :

**Code : Java**

```
import javax.xml.transform.stream.StreamResult;
```

Avant d'exécuter notre programme, il nous reste encore quelques petits détails à régler : l'écriture du prologue et le formatage de l'affichage.

Commençons par le prologue. Nous allons renseigner ses différentes propriétés via la méthode setOutputProperty() de notre transformer qui prend en paramètre le nom du paramètre et sa valeur :

**Code : Java**

```
transformer.setOutputProperty(OutputKeys.VERSION, "1.0");
transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
transformer.setOutputProperty(OutputKeys.STANDALONE, "yes");
```

Pour pouvoir utiliser les constantes de la classe OutputKeys, il convient d'importer le package suivant :

**Code : Java**

```
import javax.xml.transform.OutputKeys;
```

Si nous exécutons notre programme maintenant, tout sera écrit sur une seule ligne, ce qui n'est pas très lisible. C'est pourquoi nous allons donner quelques règles de formatage à notre transformer. En effet, ce que l'on souhaite c'est que notre document soit indenté. Chaque niveau différent de notre document XML sera alors décalé de 2 espaces :

**Code : Java**

```
transformer.setOutputProperty(OutputKeys.INDENT, "yes");
transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-
amount", "2");
```

## Le code complet

Nous venons donc de créer notre premier document XML ! Pour ceux qui en auraient besoin, vous trouverez le code complet du petit programme que nous venons d'écrire juste en dessous :

**Code : Java**

```
import java.io.File;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Comment;
import org.w3c.dom.Document;
import org.w3c.dom.Element;


public class ReadXMLFile {
    public static void main(final String[] args) {
        /*
* Etape 1 : récupération d'une instance de la classe
"DocumentBuilderFactory"
*/
 final DocumentBuilderFactory factory =
```

```
        DocumentBuilderFactory.newInstance();

         try {
             /*
        * Etape 2 : création d'un parseur
        */
             final DocumentBuilder builder = factory.newDocumentBuilder();

             /*
        * Etape 3 : création d'un Document
        */
             final Document document= builder.newDocument();

             /*
        * Etape 4 : création de l'Element racine
        */
             final Element racine = document.createElement("repertoire");
             document.appendChild(racine);

             /*
        * Etape 5 : création d'une personne
        */
             final Comment commentaire = document.createComment("John DOE");
             racine.appendChild(commentaire);

             final Element personne = document.createElement("personne");
             personne.setAttribute("sexe", "masculin");
             racine.appendChild(personne);

             /*
        * Etape 6 : création du nom et du prénom
        */
             final Element nom = document.createElement("nom");
             nom.appendChild(document.createTextNode("DOE"));

             final Element prenom = document.createElement("prenom");
             prenom.appendChild(document.createTextNode("John"));

             personne.appendChild(nom);
             personne.appendChild(prenom);

             /*
        * Etape 7 : récupération des numéros de téléphone
        */
             final Element telephones =
        document.createElement("telephones");

             final Element fixe = document.createElement("telephone");
             fixe.appendChild(document.createTextNode("01 02 03 04 05"));
             fixe.setAttribute("type", "fixe");

             final Element portable = document.createElement("telephone");
             portable.appendChild(document.createTextNode("06 07 08 09
        10"));
             portable.setAttribute("type", "portable");

             telephones.appendChild(fixe);
             telephones.appendChild(portable);
             personne.appendChild(telephones);

             /*
        * Etape 8 : affichage
        */
             final TransformerFactory transformerFactory =
        TransformerFactory.newInstance();
             final Transformer transformer =
        transformerFactory.newTransformer();
             final DOMSource source = new DOMSource(document);
             final StreamResult sortie = new StreamResult(new
        File("F:\\file.xml"));
```

```
        //final StreamResult result = new StreamResult(System.out);

        //prologue
        transformer.setOutputProperty(OutputKeys.VERSION, "1.0");
        transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");
        transformer.setOutputProperty(OutputKeys.STANDALONE, "yes");

        //formatage
        transformer.setOutputProperty(OutputKeys.INDENT, "yes");

    transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-
    amount", "2");

        //sortie
        transformer.transform(source, sortie);
     }
     catch (final ParserConfigurationException e) {
        e.printStackTrace();
     }
     catch (TransformerConfigurationException e) {
        e.printStackTrace();
     }
     catch (TransformerException e) {
        e.printStackTrace();
     }
       }
    }
```

#  XPath : Introduction à l'API

Après avoir découvert l'API DOM durant les 2 chapitres précédents, je vous propose maintenant de nous attaquer à une seconde API également très utilisée par les utilisateurs de données au format XML : l'API XPath.

## Qu'est-ce que l'API XPath ?

### La petite histoire de XPath

**XPath** est une technologie qui permet d'extraire des informations (éléments, attributs, commentaires, etc...) d'un document XML via l'écriture de d'expressions dont la syntaxe rappelle les expressions rationnelles utilisées dans d'autres langages.

Tout comme DOM, XPath est un standard du W3C et ce depuis sa première version en 1999. A moment où j'écris ces lignes, la technologies en est à sa deuxième version.

Si XPath n'est pas un langage de programmation en soit, cette technologie fournit tout un vocabulaire pour écrire des expressions permettant d'accéder directement aux informations souhaitées sans avoir à parcourir tout l'arbre XML.

### Un peu de vocabulaire

Avant d'étudier de manière plus approfondie comment écrire des expressions XPaths, il convient de revenir sur quelques notions de vocabulaires qui seront indispensables pour bien comprendre la suite du cours.

Pour se faire, je vous propose de reprendre un document XML que nous avons déjà vu plusieurs fois dans ce cours et l'utiliser pour illustrer les notions que nous allons voir :

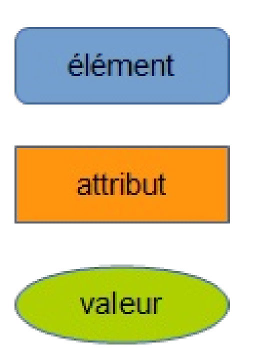
**Code : XML**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
    </personne>
</repertoire>
```

Reprenons également une illustration de son arbre :

repertoire
personne
sexe
nom
DOE
prenom
John
telephones
telephone
01 02 03 04 05
fixe
telephone
06 07 08 09 10
portable

Pour rappel, voici la légende :




### *Parent*

Le **parent** d'un nœud est le nœud qui est directement au dessus de lui d'un point de vue hiérarchique. Chaque nœud a au plus un parent.

Par exemple, le nœud **repertoire** est le parent du nœud **personne** qui est lui même le parent des noeuds **nom**, **prenom** et **telephones**.

### *Enfant*

Un nœud a pour **enfants** tous les noeuds situés un niveau en dessous dans la hiérarchie. Un nœud peut donc avoir une infinité d'enfants.

Par exemple, le nœud **repertoire** a pour enfant le nœud **personne** qui a lui même plusieurs enfants : les noeuds **nom**, **prenom** et

**telephones**.

### *Descendant*

Un nœud a pour **descendants** tous les noeuds situés en dessous dans la hiérarchie. Un nœud peut donc avoir une infinité de descendants.

Par exemple, le nœud **repertoire** a pour descendants les nœuds **personne**, **nom**, **prenom** et **telephones**.

### *Ancêtre*

Un nœud a pour **ancêtres** tous les noeuds situés en dessus dans la hiérarchie. Un nœud peut donc avoir plusieurs ancêtres.

Par exemple, le nœud **telephones** a pour ancêtres les nœuds **personne** et **repertoire**.

### *Frère*

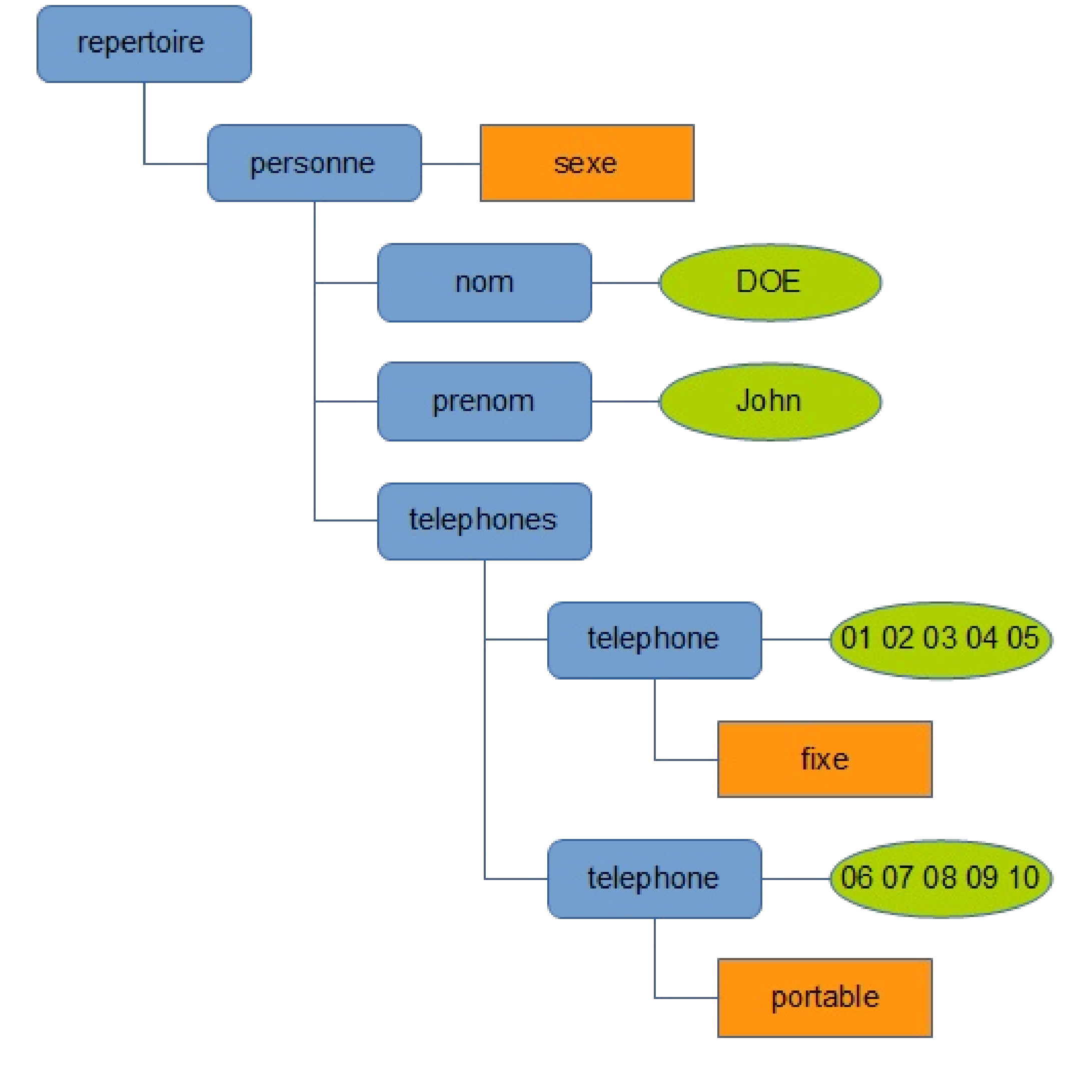
Un nœud a pour **frères** tous les noeuds situés en même niveau dans la hiérarchie. Un nœud peut donc avoir une infinité de frères.

Par exemple, le nœud **nom** a pour frères les nœuds **prenom** et **telephones**.

## Chemin relatif et chemin absolu

Vous l'aurez compris avec le chapitre précédent, XPath est une technologie qui permet d'extraire des informations d'un document XML via l'écriture de d'expressions. Concrètement ces expressions consistent à décrire le chemin emprunté dans l'arbre XML pour attendre les données qui nous intéressent.

Reprenons le schéma utilisé jusqu'ici pour illustrer le principe :



Si je veux récupérer par exemple le numéro de fixe voici le chemin à parcourir :

- étape 1 : nœud "**repertoire**" ;
- étape 2 : descendre au nœud enfant "**personne**" ;
- étape 3 : descendre au nœud enfant "**telephones**" ;
- étape 4 : descendre au nœud enfant "**telephone**" dont l'attribut est "**fixe**" ;

Sans rentrer dans les détails, l'expression XPath correspondante ressemblera à quelque chose comme ça :

**Code : XPath**

```
/étape1/étape2/étape3/étape4
```

Si le principe est toujours le même, il est possible d'exprimer vos chemins de 2 manières :

- un chemin relatif ;
- un chemin absolu.

## Les chemins absolus

**Le chemin absolu** est le type de chemin que nous avons utilisé dans notre exemple. Le nœud de départ est toujours la racine de l'arbre XML.

Une expression XPath utilisant un chemin absolu est facilement identifiable car elle commence par le caractère "/".

Bien que nous ayons déjà vu un exemple, je vous propose d'illustrer cette définition par un nouvel exemple dans lequel nous allons récupérer le prénom de la personne décrite dans notre arbre XML :

- étape 1 : nœud "**repertoire**" ;
- étape 2 : descendre au nœud enfant "**personne**" ;
- étape 3 : descendre au nœud enfant "**prenom**" ;

L'expression XPath correspondante ressemblera alors à ça :

**Code : XPath**

```
/étape1/étape2/étape3
```

## Les chemins relatifs

Si un chemin absolu est un chemin dont le nœud de départ est toujours la racine de l'arbre XML, **un chemin relatif** accepte quant à lui n'importe quel nœud de l'arbre XML comme point de départ.

Une expression XPath utilisant un chemin relatif est facilement identifiable car elle ne commence pas par le caractère "/".

Comme pour les chemins absolus, je vous propose d'illustrer cette nouvelle définition par un exemple dans lequel nous allons récupérer le prénom de la personne. Dans cet exemple, notre point de départ sera le nœud décrivant le numéro de téléphone portable de John DOE :

- étape 1 : nœud "**telephone**" dont l'attribut est "**portable**" ;
- étape 2 : remonter au nœud parent "**telephones**" ;

- étape 3 : aller nœud frère "**prenom**" ;

L'expression XPath correspondante ressemblera alors à ça :

**Code : XPath**

```
étape1/étape2/étape3
```

Dans ce premier chapitre, nous avons pu découvrir ce qu'est la technologie XPath et nous armer correctement pour la suite en comprenant le différence entre un chemin absolu et un chemin relatif sans oublier de revenir le vocabulaire indispensable à la compréhension de ce chapitre.

#  XPath : Localiser les données

Si dans le chapitre précédent nous avons fait la connaissance de XPath, dans ce nouveau chapitre les choses sérieuses commencent !

## Dissection d'une étape

Dans le chapitre précédent nous avons vu qu'une expression XPath est en réalité une succession d'étapes. Nous allons maintenant nous intéresser de plus près à ce qu'une étape.

Une étape est décrite par 3 éléments :

- un axe ;
- un nœud ou un type de nœud ;
- un ou plusieurs prédicats (facultatif).

Avant de voir en détail les valeurs possibles pour ces 3 éléments, je vous propose de revenir très rapidement sur leur rôle.

### *L'axe*

L'**axe** va nous permettre de définir le sens de la recherche, comme par si l'on souhaite se diriger vers un nœud enfant ou au contraire remonter vers un nœud parent voir un ancêtre.

### *Le nœud*

Ce second élément va nous permettre d'affiner notre recherche en indiquant explicitement le **nom d'un nœud** ou le **type de nœud** dont les informations nous intéressent.

### *Les prédicats*

Comme précisé un peu plus haut, ce dernier élément est facultatif. Les **prédicats** dont le nombre n'est pas limité agissent comme un filtre et vont nous permettre de gagner en précision lors de nos recherches. Ainsi, grâce aux **prédicats**, il sera par exemple possible de sélectionner les informations à une position précise.

Maintenant que nous connaissons comment former une étape, il nous reste à voir la syntaxe permettant de les ordonner et ainsi écrire une étape compatible avec XPath :

**Code : XPath**

```
axe::nœud[predicat][predicat]...[predicat]
```

## Les axes

Comme nous l'avons vu dans la partie précédente, un axe est le premier élément formant une étape. Son rôle est de définir le sens de la recherche. Bien évidemment, le choix du sens est structuré par un vocabulaire précis que nous allons étudier maintenant.

## Le tableau récapitulatif

| Nom de l'axe | Description |
|---|---|
| ancestor | oriente la recherche vers les **ancêtres** du nœud courant |
| ancestor-or-self | oriente la recherche vers le **nœud courant** et ses **ancêtres** |
| attribute | oriente la recherche vers les **attributs** du nœud courant |
| child | oriente la recherche vers les **enfants** du nœud courant |

| | |
|---|---|
| descendant | oriente la recherche vers les **descendants** du nœud courant |
| descendant-or-self | oriente la recherche vers le **nœud courant** et ses **descendants** |
| following | oriente la recherche vers les **nœuds suivant** le nœud courant |
| following-sibling | oriente la recherche vers les **frères suivants** du nœud courant |
| parent | oriente la recherche vers le **père** du nœud courant |
| preceding | oriente la recherche vers les **nœuds précédant** le nœud courant |
| preceding-sibling | oriente la recherche vers les **frères précédents** du nœud courant |
| self | oriente la recherche vers le **nœud courant** |



Pour votre culture générale, sachez qu'il existe également un axe nommé **namespace** qui permet d'orienter la recherche vers un espace de nom. Je l'ai volontairement retiré du tableau récapitulatif car nous ne l'utiliserons pas dans le cadre de ce tutoriel.

## Quelques abréviations

Tout au long de notre découverte des axes, des tests de nœuds et des prédicats, nous allons découvrir qu'il est possible d'utiliser des abréviations afin de rendre la syntaxe de nos expressions XPath plus claire et concise.

### *L'axe child*

Pour les axes, il existe une abréviation possible et elle concerne l'axe **child**. En réalité, lorsque l'on souhaite orienter la recherche vers l'axe child, ce n'est pas nécessaire de le préciser. Il s'agit de l'axe par défaut.

## Les tests de nœuds

Nous venons donc de voir les différentes valeurs possibles pour sélectionner un axe. Il est donc maintenant temps d'attaquer le second élément composant une étape : le **nom** d'un nœud ou le **type** de nœud.

## Le tableau récapitulatif

| Nom | Description |
|---|---|
| nom du nœud | oriente la recherche vers le nœud dont le nom a explicitement été spécifié |
| * | oriente la recherche vers **tous les nœuds** |
| node() | oriente la recherche vers **tous les types de nœuds** (éléments, commentaires, attributs, etc.) |
| text() | oriente la recherche vers **les nœuds de type texte** |
| comment() | oriente la recherche vers **les nœuds de type commentaire** |



A noter qu'il existe également d'autres valeurs possibles comme par exemple **processing-instruction()**. Je l'ai volontairement retiré du tableau récapitulatif car nous ne l'utiliserons pas dans le cadre de ce tutoriel.

## Quelques exemples

Puisque les prédicats sont facultatifs dans les expressions XPath, je vous propre de voir d'ores et déjà quelques exemples. Pour les exemples, nous allons nous appuyer sur le document XML suivant que nous avons déjà utilisé :

**Code : XML**

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <adresse>
            <numero>7</numero>
            <voie type="impasse">impasse du chemin</voie>
            <codePostal>75015</codePostal>
            <ville>PARIS</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
        <emails>
            <email type="personnel">john.doe@wanadoo.fr</email>
            <email type="professionnel">john.doe@societe.com</email>
        </emails>
    </personne>
</repertoire>
```

### *Les chemins absolus*

Nous allons débuter par des exemples se basant sur l'écriture dune expression utilisant un chemin absolu.

Dans notre premier exemple le but va être de récupérer **le pays de domiciliation** de John DOE. Commençons par décrire les étapes à suivre en français :

- étape 1 : descendre au nœud "**repertoire**"
- étape 2 : descendre au nœud "**personne**"
- étape 3 : descendre au nœud "**adresse**"
- étape 4 : descendre au nœud "**pays**"

Traduisons maintenant ces étapes sous la forme d'expressions XPath :

- étape 1 : child::repertoire
- étape 2 : child::personne
- étape 3 : child::adresse
- étape 4 : child::pays

Ce qui nous donne :

**Code : XPath**

```
/child::repertoire/child::personne/child::adresse/child::pays
```

Il même possible de simplifier l'écriture de cette expression. En effet, comme dis dans le chapitre sur les axes, l'axe **child** est celui par défaut, il n'est donc pas nécessaire de le préciser. Ainsi, il est possible de simplifier notre expression de la sorte :

**Code : XPath**

```
/repertoire/personne/adresse/pays
```

Maintenant que vous êtes un peu plus à l'aise avec la syntaxe de XPath, je vous propose de voir un exemple un peu plus

exotique. Ainsi, le but est dorénavant de trouver l'expression XPath permettant de trouver tous les commentaires de notre document XML.

Dans ce nouvel exemple, une seule étape est en réalité nécessaire et consiste à sélectionner tous les descendants du nœud **racine** qui sont des commentaires.

Tentons maintenant de traduire cette étape sous la forme d'expressions XPath :

- on sélectionne tous les descendants avec l'expression **descendant** ;
- puis on filtre les commentaires avec l'expression **comment()**

Ce qui nous donne :

**Code : XPath**

```
/descendant::comment()
```

### *Les chemins relatifs*

Après avoir vu quelques exemples d'expressions XPath utilisant des chemins absolus, je vous propose de voir un exemple d'une expression utilisant un chemin relatif. Dans cet exemple, notre point de départ sera le nœud "**telephones**". Une fois de plus, le but va être de récupérer **le pays de domiciliation** de John DOE. Commençons par décrire les étapes à suivre en français :

- étape 1 : remonter au nœud frère "**adresse**"
- étape 2 : descendre au nœud "**pays**"

Traduisons maintenant ces étapes sous la forme d'expressions XPath :

- étape 1 : preceding-sibling::adresse
- étape 2 : pays

Ce qui nous donne :

**Code : XPath**

```
preceding-sibling::adresse/pays
```

## Quelques abréviations

Tout comme pour les axes, il existe quelques abréviations dont je vous conseille d'abuser afin de rendre vos expressions XPath plus lisibles et légères.

### *L'expression /descendant-or-self::node()/*

Dans nos expressions XPath, il est possible de remplacer l'expression "**/descendant-or-self::node()/**" par "//".

Ainsi, l'expression :

**Code : XML**

```
/descendant-or-self::node()/pays
```

peut être simplifiée par :

**Code : XML**

```
//pays
```

### *L'expression self::node()*

Notre deuxième abréviation va nous permettre de remplacer l'expression "**/self::node()/**" par "**.**".

Ainsi, l'expression :

**Code : XML**

```
/repertoire/personne/self::node()
```

peut être simplifiée par :

**Code : XML**

```
/repertoire/personne/.
```

### *L'expression parent::node()*

Finalement, notre dernière abréviation va nous permettre de remplacer l'expression "**/parent::node()/**" par "**..**".

## Les prédicats

Nous venons donc de voir 2 des 3 éléments formant une étape dans une expression XPath. Dans ce chapitre nous allons donc aborder l'élément manquant : **les prédicats**.

## Le tableau récapitulatif

| Nom du prédicat | Description |
|---|---|
| attribute | permet d'affiner la recherche en fonction d'un attribut |
| count() | permet de compter le nombre de nœuds |
| last() | permet de sélectionner le dernier nœud d'une liste |
| position() | permet d'affiner la recherche en fonction de la position d'un nœud |


A noter qu'il existe également d'autres valeurs possibles comme par exemple **name()**, **id()** ou encore **string-length()**. Je les ai volontairement retirées du tableau récapitulatif car nous ne l'utiliserons pas dans le cadre de ce tutoriel.


Un prédicat peut également contenir une expression XPath correspondant à une étape. Nous verrons notamment ce cas dans le TP.

## Quelques exemples

Pour les exemples, nous allons nous appuyer toujours sur le même document XML :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <adresse>
            <numero>7</numero>
            <voie type="impasse">impasse du chemin</voie>
            <codePostal>75015</codePostal>
            <ville>PARIS</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
        <emails>
            <email type="personnel">john.doe@wanadoo.fr</email>
            <email type="professionnel">john.doe@societe.com</email>
        </emails>
    </personne>
</repertoire>
```

### *Premier exemple*

Dans notre premier exemple le but va être de récupérer le nœud contenant **le numéro de téléphone fixe** de John DOE. Bien évidemment, il existe plusieurs façon d'y arriver. Je vous propose d'utiliser celle qui pousse le moins à réfléchir : nous allons sélectionner tous les descendants du nœud racine et filtrer sur la valeur de l'attribut **type**. Ce qui nous donne :

**Code : XPath**

```
/descendant::*[attribute::type="fixe"]
```

Bien évidemment cette méthode est à prescrire car elle peut avoir de nombreux effets de bord. Une autre manière de faire est donc de préciser le chemin complet :

**Code : XPath**

```
/repertoire/personne/telephones/telephone[attribute::type="fixe"]
```

Terminons ce premier exemple en sélectionnant les numéros de téléphones qui ne sont pas des numéros de téléphones fixes. Une fois de plus, il existe plusieurs manières de faire. La première, qui à priori la plus simple, consiste à remplacer dans notre expression précédente l'opérateur d'égalité **"="** par l'opérateur de non égalité "**!=**" :

**Code : XPath**

```
/repertoire/personne/telephones/telephone[attribute::type!="fixe"]
```

Une autre méthode consiste à utiliser la fonction **not()** :

**Code : XPath**

```
/repertoire/personne/telephones/telephone[not(attribute::type="fixe")]
```

A noter que la double négation nous fait revenir à notre point de départ. En effet, les 2 expressions suivantes sont équivalentes :

**Code : XPath**

```
/repertoire/personne/telephones/telephone[not(attribute::type!="fixe")]
```

**Code : XPath**

```
/repertoire/personne/telephones/telephone[attribute::type="fixe"]
```

### *Deuxième exemple*

Après avoir manipulé les attributs, je vous propose de manipuler les positions. Ainsi, notre deuxième exemple consiste à sélectionner **le premier numéro de téléphone** de John DOE. Commençons par détailler les étapes en français :

- étape 1 : descendre au nœud "**repertoire**"
- étape 2 : descendre au nœud "**personne**"
- étape 3 : descendre au nœud "**telephones**"
- étape 4 : sélectionner le premier nœud "**telephone**"

Traduisons maintenant ces étapes sous la forme d'expressions XPath :

- étape 1 : repertoire
- étape 2 : personne
- étape 3 : telephones
- étape 4 : telephone[position()=1]

Ce qui nous donne :

**Code : XML**

```
/repertoire/personne/telephones/telephone[position()=1]
```

Si l'on souhaite maintenant sélectionner le dernier nœud téléphone de la liste, on modifiera l'expression de la manière suivante :

**Code : XML**

```
/repertoire/personne/telephones/telephone[last()]
```

## Quelques abréviations

Comme pour les axes, nous n'allons voir ici qu'une seule abréviation et elle concerne le prédicat **attribute** qu'il est possible de remplacer par le symbole "@". Ainsi, l'expression :

**Code : XPath**

```
/repertoire/personne/telephones/telephone[attribute::type="fixe"]
```

devient :

**Code : XPath**

```
/repertoire/personne/telephones/telephone[@type="fixe"]
```

## Un exemple avec EditiX

Afin de terminer ce chapitre, je vous propose de voir comment exécuter une expression XPath avec EditiX.

## Le document XML

Commencer par créer dans EditiX un document XML contenant les informations suivantes :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

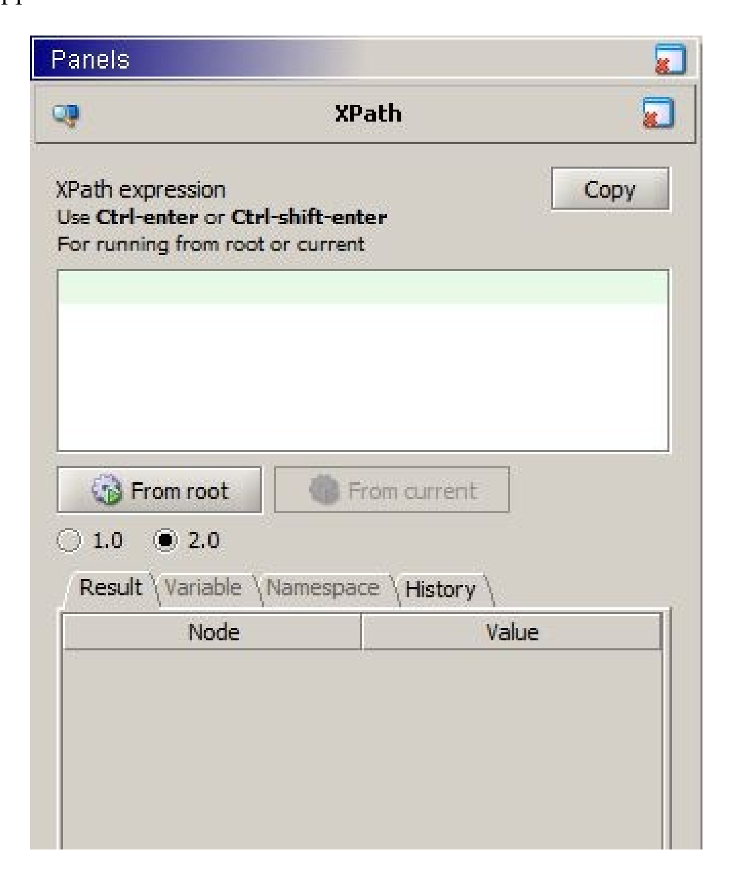
<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <adresse>
            <numero>7</numero>
            <voie type="impasse">impasse du chemin</voie>
            <codePostal>75015</codePostal>
            <ville>PARIS</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
        <emails>
            <email type="personnel">john.doe@wanadoo.fr</email>
            <email type="professionnel">john.doe@societe.com</email>
        </emails>
    </personne>

    <!-- Marie POPPINS -->
    <personne sexe="feminin">
        <nom>POPPINS</nom>
        <prenom>Marie</prenom>
        <adresse>
            <numero>28</numero>
            <voie type="avenue">avenue de la république</voie>
            <codePostal>13005</codePostal>
            <ville>MARSEILLE</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="bureau">04 05 06 07 08</telephone>
        </telephones>
        <emails>
            <email type="professionnel">contact@poppins.fr</email>
        </emails>
    </personne>
</repertoire>
```

## La vue XPath

Afin de pouvoir exécuter des expressions XPath, nous allons devoir afficher la vue dédiée au sein de EditiX. Pour se faire, vous pouvez sélectionner dans la barre de menu **XML** puis **XPath view** ou encore utiliser le raccourci clavier Ctrl + Shift + 4.

La fenêtre suivante doit alors apparaître :




Comme vous pouvez le constater, cette vue se compose de plusieurs éléments :

- un champ dans lequel toutes nos expressions seront écrites ;
- 2 boutons permettant de choisir si notre expression utilise un chemin relatif ou absolu ;
- une puce permettant de choisir la version de XPath à utiliser (prenez l'habitude de travailler avec la version 2) ;
- des onglets permettant entre autres d'afficher les informations sélectionnées par nos expressions.

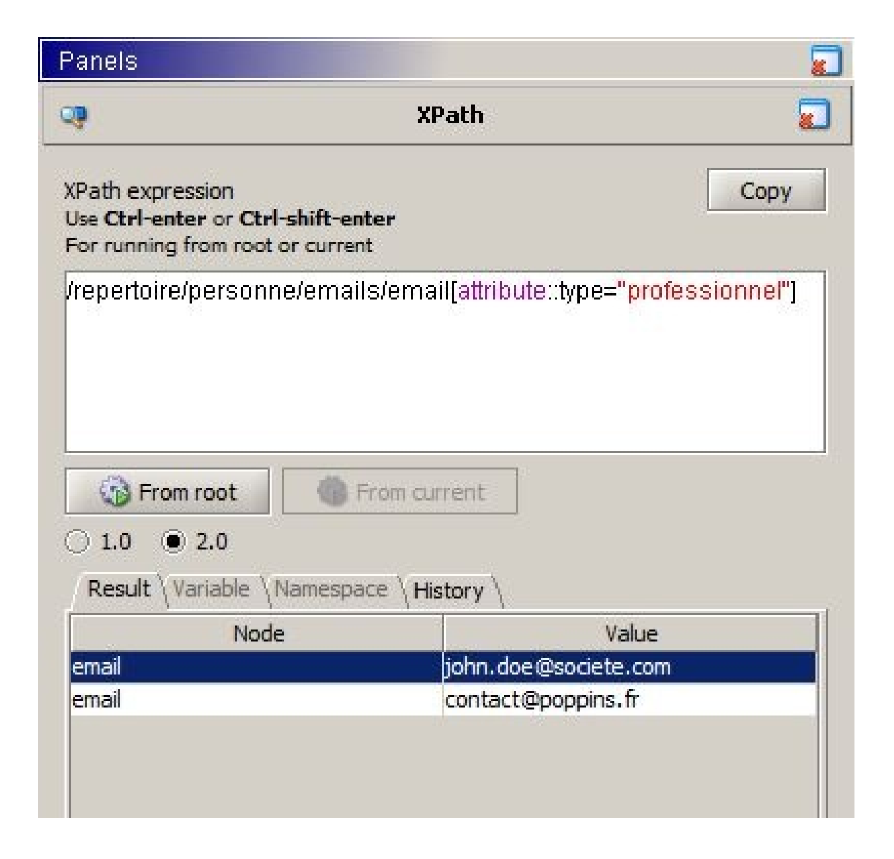
## Exécuter une requête

Dans cet ultime exemple, nous allons sélectionner les nœuds contenant des adresses e-mails professionnelle grâce à l'expression suivante :

**Code : XPath**

```
/repertoire/personne/emails/email[attribute::type="professionnel"]
```

En théorie, nous devrions avoir 2 noeuds de sélectionnés. Vérifions tout de suite :




Le résultat est bien celui souhaité !

#  TP : des expressions XPath dans un répertoire

Votre apprentissage des expressions XPath arrive à sa fin. Pour bien terminer votre apprentissage, rien de mieux qu'un peu de pratique à travers un petit TP.

Le principe du TP est simple : écrire les expressions XPath permettant de récupérer les informations demandées.

## L'énoncé

## Le document XML

Une fois de plus, c'est avec un répertoire téléphonique que nous allons travailler.

Voici les informations que l'on connaît pour chaque personne :

- son sexe (homme ou femme) ;
- son nom ;
- son prénom ;
- son adresse ;
- un ou plusieurs numéros de téléphone (téléphone portable, fixe, bureau, etc.) ;
- zero ou plusieurs adresses e-mail (adresse personnelle, professionnelle, etc.).

Voici maintenant le document XML qui va nous servir de support :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<repertoire>
    <!-- John DOE -->
    <personne sexe="masculin">
        <nom>DOE</nom>
        <prenom>John</prenom>
        <adresse>
            <numero>7</numero>
            <voie type="impasse">impasse du chemin</voie>
            <codePostal>75015</codePostal>
            <ville>PARIS</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
        </telephones>
        <emails>
            <email type="personnel">john.doe@wanadoo.fr</email>
            <email type="professionnel">john.doe@societe.com</email>
        </emails>
    </personne>

    <!-- Marie POPPINS -->
    <personne sexe="feminin">
        <nom>POPPINS</nom>
        <prenom>Marie</prenom>
        <adresse>
            <numero>28</numero>
            <voie type="avenue">avenue de la république</voie>
            <codePostal>13005</codePostal>
            <ville>MARSEILLE</ville>
            <pays>FRANCE</pays>
        </adresse>
        <telephones>
            <telephone type="professionnel">04 05 06 07
08</telephone>
        </telephones>
        <emails>
            <email type="professionnel">contact@poppins.fr</email>
```

```
            </emails>
        </personne>

     <!-- Batte MAN -->
        <personne sexe="masculin">
            <nom>MAN</nom>
            <prenom>Batte</prenom>
            <adresse>
                <numero>24</numero>
                <voie type="avenue">impasse des héros</voie>
                <codePostal>11004</codePostal>
                <ville>GOTHAM CITY</ville>
                <pays>USA</pays>
            </adresse>
            <telephones>
                <telephone type="professionnel">01 03 05 07
09</telephone>
            </telephones>
        </personne>
    </repertoire>
```

## Les expressions à écrire

Voici donc la liste des expressions XPath à écrire :

- sélectionner tous les nœuds descendants du deuxième nœuds "**personne**" ;
- sélectionner le nœud "**personne**" correspondant au individu ayant au moins 2 numéros de téléphone ;
- sélectionner tous les nœud "**personne**"
- sélectionner le deuxième nœud "**personne**" dont le pays de domiciliation est la **France** ;
- sélectionner le tous les nœuds "**personne**" de sexe **masculin** le pays de domiciliation est les **Etats-Unis** ;

C'est à vous de jouer !

## Une solution

Comme à chaque fois, je vous fais part de ma solution !

## Expression n°1

Le but de cette première expression était de sélectionner tous les nœuds descendants du deuxième nœuds "**personne**" :

**Code : XPath**

```
/repertoire/personne[position()=2]/descendant::*
```

## Expression n°2

Le but de cette expression était de sélectionner le nœud "**personne**" correspondant au individu ayant au moins 2 numéros de téléphone :

**Code : XPath**

```
/repertoire/personne[count(telephones/telephone) > 1]
```

## Expression n°3

Le but de cette troisième expression était de sélectionner tous les nœud "**personne**" :

**Code : XPath**

```
/repertoire/personne
```

ou encore :

**Code : XPath**

```
//personne
```

## Expression n°4

Le but de cette expression était de sélectionner le deuxième nœud "**personne**" dont le pays de domiciliation est la **France** :

**Code : XPath**

```
/repertoire/personne[adresse/pays="FRANCE"][position()=2]
```

## Expression n°5

Le but de la dernière expression était de sélectionner tous les nœuds "**personne**" de sexe **masculin** le pays de domiciliation est les Etats-Unis :

**Code : XPath**

```
/repertoire/personne[@sexe="masculin"][adresse/pays="USA"]
```

# Partie 4 : Annexes

##  Les espaces de noms

Dans ce chapitre, nous allons revenir sur un concept important en XML : **les espaces de noms**.

### Définition

#### Définition d'un espace de noms

Lorsque l'on écrit un document XML, on utilise ce que l'on appelle un vocabulaire. Par exemple, dans les différents TP, nous avons travaillé avec des répertoires téléphoniques dans lesquels chaque personne a :

- une identité (un nom et un prénom) ;
- une adresse ;
- des numéros de téléphones ;
- des adresses e-mails ;
- etc.

A travers cette description, nous avons avons définit le vocabulaire d'une personne.

Dans notre cas, notre vocabulaire n'est pas forcément réutilisable en l'état. Et pourtant, en informatique, on aime bien réutiliser ce que l'on fait, ne pas toujours repartir à zéro.

Il existe un certain nombre de vocabulaires qui ont fait leurs preuves et qui ont été mis à dispositions des développeurs afin qu'ils puissent être réutilisés. Bien que nous y reviendrons un peu plus tard, nous pouvons citer quelques vocabulaires :

- le vocabulaire permettant de décrire une page xHTML ;
- le vocabulaire permettant de décrire un Schéma XML ;
- le vocabulaire permettant de décrire des documents techniques ;
- etc.

#### Identifier un espace de noms

Un espace de noms est identifié par une URI (Uniform Resource Identifier) qui permet de l'identifier de manière unique. Bien que l'on distingue 2 types d'URI, à savoir les URL (Uniform Resource Locator) et les URN (Uniform Resource Name), dans la majorité des cas, c'est une URL qui est utilisée.

Pour rappel, une URL permet d'identifier de manière unique une ressource, dans notre cas un vocabulaire, sur un réseau.

Voyons quelques exemples d'URL permettant d'identifier des vocabulaires et donc des espaces de noms sur le réseau internet :

- xhtml : http://www.w3.org/1999/xhtml
- Schéma XML : http://www.w3.org/2001/XMLSchema
- DocBook : http://docbook.org/ns/docbook

### Utilisation d'un espace de noms

#### Les espaces de noms par défaut

La déclaration d'un espace de noms par défaut se fait dans le premier élément qui utilise le vocabulaire, grâce au mot clef **xmlns** comme **XML namespace**.

**Code : XML**

```
xmlns="mon_uri"
```

Illustrons alors la déclaration et l'utilisation d'un espace de noms par défaut via l'exemple d'un document xHTLM :

**Code : XML**

```
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Titre du document</title>
    </head>
    <body>
        <p>
            <img src="mon_image.png" alt="ma super image" />
            <br/>
            <a href="mon_lien">Mon super lien !</a>
        </p>
    </body>
</html>
```

Tous les éléments utilisés dans ce document XML comme `<head />`, `<title />`, `<body />`, etc, font partie du vocabulaire d'une page xHTML. C'est grâce à la déclaration de l'espace de nom dans l'élément `<html />` que nous pouvons utiliser les différents éléments du vocabulaire tout en respectant les règles qui régissent leurs imbrications les uns par rapport aux autres.

## Les espaces de noms avec préfixe

Utiliser un espace de noms par défaut à certaines limites. En effet, il par exemple impossible d'utiliser au sein d'un même document 2 espaces de nom par défaut qui auraient un mot de vocabulaire en commun. En effet, on obtiendrait alors une ambiguïté.

Tout problème a bien évidemment une solution Ainsi, pour résoudre ce problème, nous allons utiliser des préfixes avec nos espaces de noms.

Tout comme pour un espace de noms par défaut, la déclaration d'un espace de nom avec préfixe se fait dans le premier élément qui utilise le vocabulaire, grâce au mot clef **xmlns:prefixe**.

**Code : XML**

```
xmlns:prefixe="mon_uri"
```

Lorsqu'un espace de noms est déclaré avec un préfixe, tous les éléments qui appartiennent au vocabulaire et donc à l'espace de nom doivent être précédés par ce préfixe :

**Code : XML**

```
<prefixe:element />
```

Afin d'illustrer cette nouvelle notion, reprenons la page xHTML que nous avons avons écrit plus haut, et utilisons cette fois-ci un préfixe :

**Code : XML**

```
<http:html xmlns:http="http://www.w3.org/1999/xhtml">
    <http:head>
        <http:title>Titre du document</http:title>
    </http:head>
    <http:body>
        <http:p>
            <http:img src="mon_image.png" alt="ma super image" />
```

```
                <http:br/>
                <http:a href="mon_lien">Mon super lien !</http:a>
            </http:p>
        </http:body>
    </http:html>
```

## La portée d'un espace de noms

Pour terminer ce chapitre, il me semble intéressant de revenir sur la notion de portée d'un espace de noms. En effet, suivant la façon dont il est déclaré, un espace de noms n'est pas accessible partout dans un document, il n'est donc pas possible d'utiliser tous les mots de vocabulaire partout dans un document XML.

La règle qui régie la portée d'un espace de noms est assez simple : un espace de noms est utilisable tant que l'élément qui le déclare n'est pas refermé.

Une fois de plus, je vous propose d'illustrer cette notion via un exemple :

**Code : XML**

```
<!-- il est possible d'utiliser l'espace de noms http -->
<http:html xmlns:http="http://www.w3.org/1999/xhtml">
    <http:head>
        <http:title>Titre du document</http:title>
    </http:head>
    <http:body>
        <http:p>
           <!-- il est possible d'utiliser l'espace de noms ml -->
           <ml:math xmlns:ml="http://www.w3.org/1998/Math/MathML">
               <ml:matrix>
                   <ml:matrixrow>
                       <ml:cn>0</ml:cn>
                       <ml:cn>1</ml:cn>
                       <ml:cn>0</ml:cn>
                   </ml:matrixrow>
                   <ml:matrixrow>
                       <ml:cn>0</ml:cn>
                       <ml:cn>0</ml:cn>
                       <ml:cn>1</ml:cn>
                   </ml:matrixrow>
               </ml:matrix>
           </ml:math>
           <!-- il n'est plus possible d'utiliser l'espace de noms
ml -->
        </http:p>
    </http:body>
</http:html>
<!-- il n'est plus possible d'utiliser l'espace de noms http -->
```

## Quelques espaces de noms utilisés régulièrement

Afin de conclure ce chapitre, je vous propose de revenir sur quelques espaces de noms connus, régulièrement utilisés par les développeurs. Bien évidemment cette liste n'est pas exhaustive, il est existe un très grand nombre !

## DocBook

**DocBook** permet de décrire des documents techniques comme des livres, des articles, etc.

Cet espace de noms est identifié par l'URI http://docbook.org/ns/docbook

## MathML

**MathML** est une spécification du W3C qui permet d'afficher éléments mathématiques diverses et variés comme des additions, des soustractions, des matrices, etc.

Cet espace de noms est identifié par l'URI http://www.w3.org/1998/Math/MathML

## Schéma XML

Il s'agit d'une spécification du W3C qui permet de décrire des Schémas XML.

Cet espace de noms est identifié par l'URI http://www.w3.org/2001/XMLSchema

## SVG

**SVG** pour **Scalable Vector Graphics** est une spécification du W3C qui permet de décrire des images vectorielles.

Cet espace de noms est identifié par l'URI http://www.w3.org/2000/svg

## XLink

Il s'agit d'une spécification du W3C permettant de créer des liens entre plusieurs fichiers XML.

Cet espace de noms est identifié par l'URI http://www.w3.org/1999/xlink

## XSLT

**XSLT** pour **eXtensible Stylesheet Language Transformations** est une spécification du W3C qui permet de décrire des transformations à appliquer un document XML.

Cet espace de noms est identifié par l'URI http://www.w3.org/1999/XSL/Transform

#  Mettez en forme vos documents XML avec CSS

Dans ce chapitre, nous n'allons pas revenir en détail sur chaque propriété du CSS. En effet, ce langage nécessite à lui tout seul l'écriture d'un tutoriel dont la taille serait conséquente. Le but est ici de découvrir ce qu'il est possible de faire avec du CSS et du XML.

## Ecrire un document CSS

### Qu'est-ce que le CSS ?

Bien que je sois certain que la majorité d'entre vous connaissent déjà grosso modo ce qu'est le CSS, je vous propose de revenir rapidement sur ce langage.

Le CSS ou Cascading Style Sheet de son nom complet, est un langage informatique à part entière permettant de styliser les documents HTML et XML. Avec le HTML, il s'agit du langage qui permet de mettre en forme et styliser les sites internet.

Nous allons donc voir comment grâce au CSS, il est possible de styliser un document XML. Par styliser, j'entends par exemple :

- écrire certains éléments en gras ;
- souligner ;
- surligner ;
- écrire certains éléments en couleur ;
- etc.

### Ou écrire le CSS ?

Vous commencez à le comprendre, en informatique, on aime bien séparer les différentes technologies dans différents fichiers. Le CSS ne fait pas exception à la règle. Ainsi, nos lignes de CSS seront écrites dans un fichier portant l'extension de fichier **.css**.

### Référencer le fichier CSS

Afin de pouvoir appliquer un style CSS à nos documents XML, il convient de lier les différents fichiers entre eux.

Cette liaison se fait dans le document XML entre le prologue et la racine, grâce à la balise suivante :

**Code : XML**

```
<?xml-stylesheet href="style.css" type="text/css" ?>
```

Prenons par exemple le document XML suivant :

**Code : XML**

```
<?xml version = "1.0" encoding="UTF-8">
<?xml-stylesheet href="personne.css" type="text/css" ?>
<personne>
    <nom>NORRIS</nom>
    <prenom>Chuck</prenom>
</personne>
```

Dans cet exemple, l'affichage du document XML sera stylisé grâce au document CSS **personne.css.**

### Syntaxe du CSS

Comme tous les langages, le CSS dispose d'une syntaxe qui lui est propre. L'idée du langage est de sélectionner une ou plusieurs éléments d'un document afin d'y appliquer un certain nombre de propriétés.

Dans ce chapitre, je ne vais pas revenir sur l'ensemble des propriétés qui existent en CSS. Cependant, je vous encourage à lire le

mémo écrit par M@teo21 dans son tutoriel sur le HTML5 et le CSS3.

### *Sélectionner une balise*

En CSS, pour sélectionner un élément particulier d'un document XML, on utilise la syntaxe suivante :

**Code : CSS**

```
balise {
    propriété1 : valeur;
    propriété2: valeur;
}
```

Illustrons cette première règle grâce à un exemple. Soit le document XML suivant représentant une liste de personnes :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="personnes.css" type="text/css" ?>
<personnes>
    <personne>
        <nom>NORRIS</nom>
        <prenom>Chuck</prenom>
    </personne>

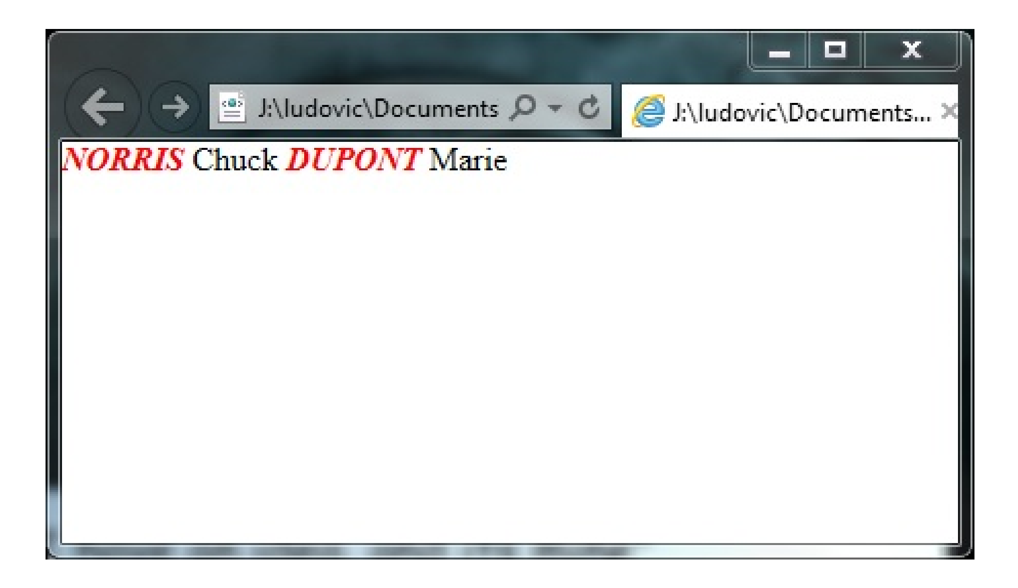
    <personne>
        <nom>DUPONT</nom>
        <prenom>Marie</prenom>
    </personne>
</personnes>
```

Si je souhaite par exemple afficher les prénoms des différentes personnes en rouge, gras et italique. Écrivons alors le contenu du fichier **personnes.css** :

**Code : CSS**

```
nom {
    color:red;
    font-weight:bold;
    font-style:italic;
}
```

Si vous ouvrez votre document XML dans un navigateur web, vous devriez alors avoir l'affichage indiqué sur la figure suivante.




Comme on le souhaitait, les noms des personnes sont bien écrits en rouge, gras et italique.

### *Sélectionner une balise particulière*

Allons un petit peu plus loin et considérons le document XML suivant, représentant encore une liste de personnes :

**Code : XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<personnes>
    <personne sexe="masculin">
        <nom>NORRIS</nom>
        <prenom>Chuck</prenom>
    </personne>

    <personne sexe="feminin">
        <nom>DUPONT</nom>
        <prenom>Marie</prenom>
    </personne>
</personnes>
```

Comment faire si je souhaite par exemple afficher le nom des hommes en bleu et celui des femmes en roses ? Pour arriver à ce résultat, il convient de sélectionner une personne en fonction de l'attribut **sexe**.

Pour sélectionner une balise particulière en fonction de la valeur d'un attribut en CSS, on utilise la syntaxe suivante :

**Code : CSS**

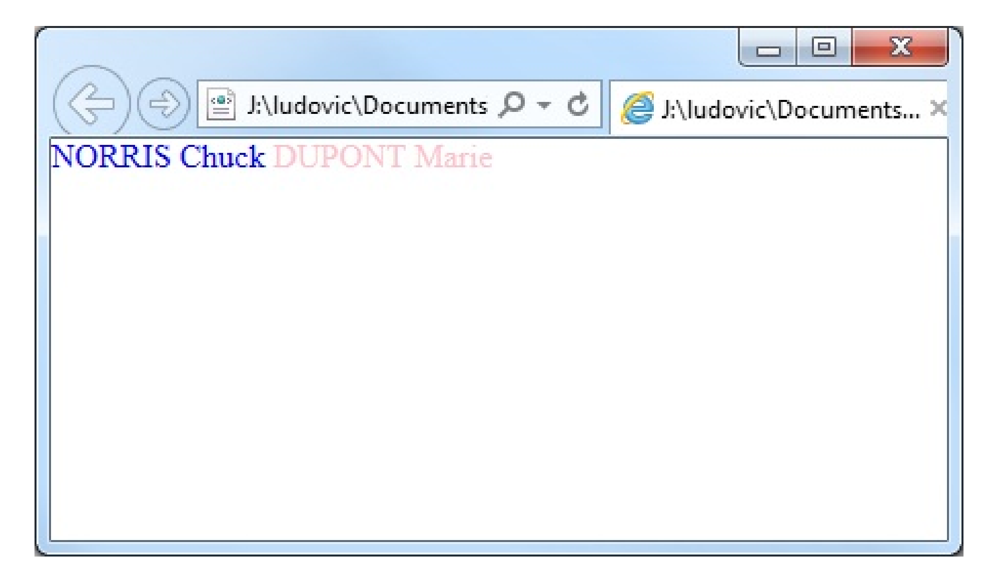
```
balise[attribut="valeur"] {
    propriété1 : valeur;
    propriété2: valeur;
}
```

Tentons alors d'appliquer le style suivant à notre document XML :

**Code : CSS**

```
personne[sexe="masculin"] { color:blue; }
personne[sexe="feminin"] { color:pink; }
```

Si l'on affiche le document XML dans un navigateur web, on obtient le résultat affiché en figure suivante.




On s'approche du résultat souhaité, mais ce n'est pas encore ça. En effet, actuellement, les noms et prénoms des personnes sont colorées, or ce que l'on souhaite nous, c'est qu'uniquement des noms des personnes soient colorés.

Il nous suffit de légèrement modifier notre feuille de style :

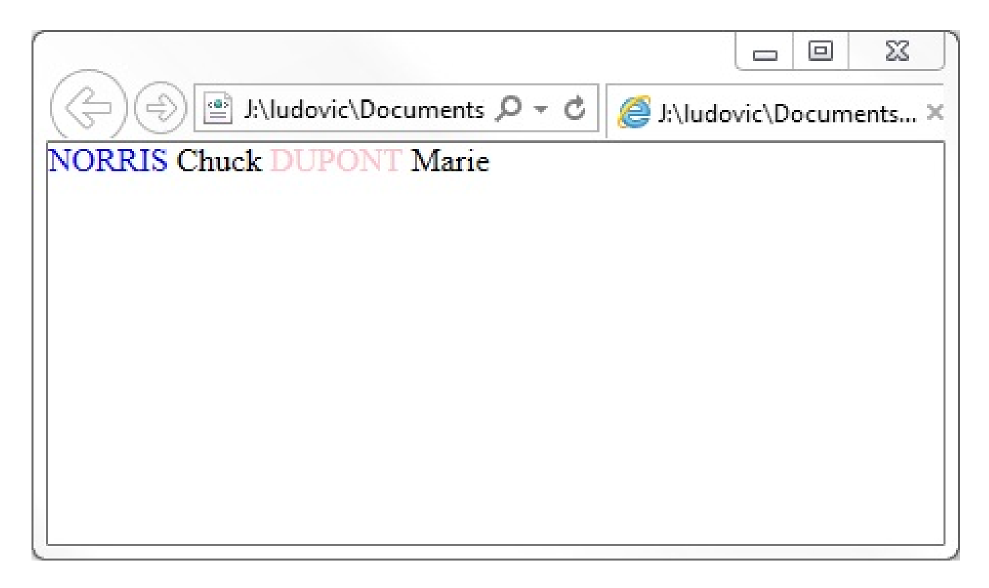
**Code : CSS**

```
personne[sexe="masculin"] nom { color:blue; }
personne[sexe="feminin"] nom { color:pink; }
```

Comme vous pouvez le constater, on a ajouté l'élément **nom** au niveau de notre sélection. En français, on pourrait traduire ces lignes de CSS par les 2 phrases suivantes :

- écrit en bleu le nom des personnes de sexe masculin ;
- écrit en rose le nom des personnes de sexe féminin ;

Finalement, si vous affichez le document XML dans un navigateur web, vous devriez avoir le résultat affiché en figure suivante, correspondant bien à nos attentes.



## Un exemple avec EditiX

Comme pour chaque technologie que nous voyons ensemble, je vous propose de voir comment procéder grâce au logiciel EditiX.

## Création du document XML

La création du document XML n'a rien de bien compliqué puisque nous l'avons déjà vu ensemble à plusieurs reprise.

Pour ceux qui ne s'en souviennent pas, vous pouvez y jeter un coup d'œil ici.

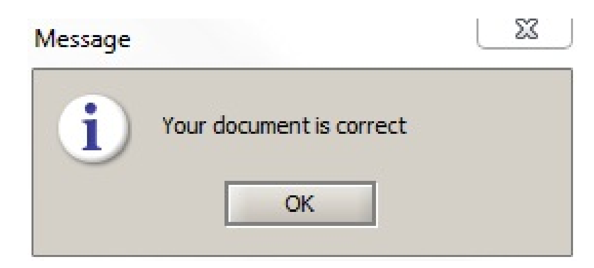
Je vous propose de reprendre pour exemple notre liste de personnes :

**Code : XML**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="personnes.css" type="text/css" ?>
<personnes>
    <personne sexe="masculin">
        <nom>NORRIS</nom>
        <prenom>Chuck</prenom>
    </personne>

    <personne sexe="feminin">
        <nom>DUPONT</nom>
        <prenom>Marie</prenom>
    </personne>
</personnes>
```

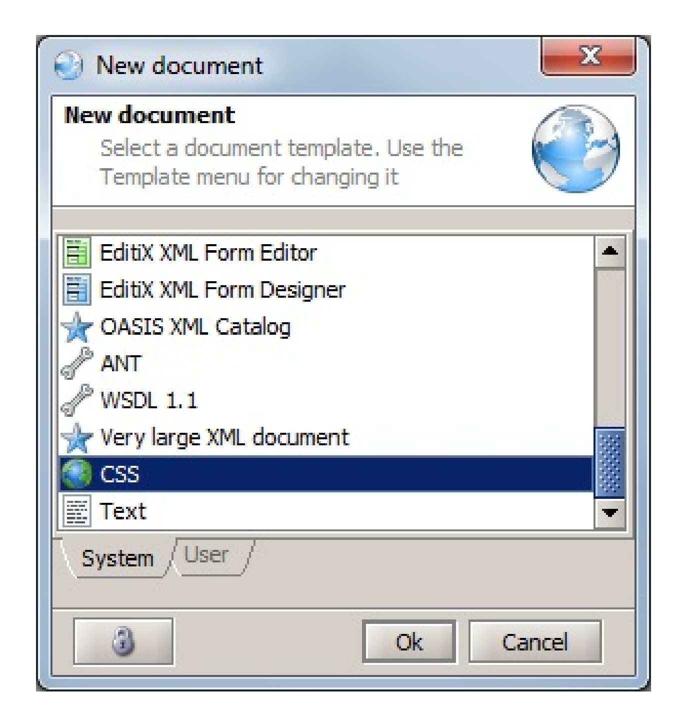
Si vous essayez de lancer la vérification du document, vous devriez normalement avoir ce message (voir la figure suivante).




## Création du document CSS

Pour créer un nouveau document, vous pouvez sélectionner dans la barre de menu **File** puis **New** ou utiliser le raccourci clavier Ctrl + N.

Dans la liste qui s'affiche, sélectionnez **CSS**, comme indiqué sur la figure suivante.




Votre document CSS n'est normalement pas vierge. Voici ce que vous devriez avoir :

**Code : CSS**

```css
/* Generated with EditiX at Sat May 11 19:46:07 CEST 2013 */
```

Replacez le contenu par notre véritable CSS :

**Code : CSS**

```css
personne[sexe="masculin"] nom { color:blue; }
personne[sexe="feminin"] nom { color:pink; }
```

Enregistrez ensuite votre document avec le nom **personnes.css** au même endroit que votre document XML.

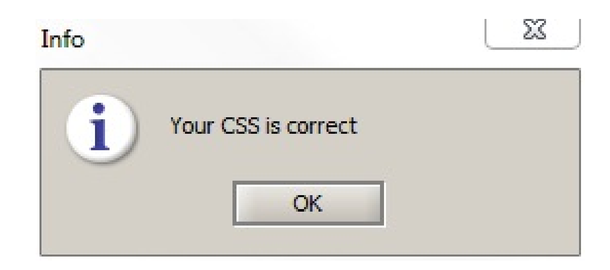
## Vérification de fichier de style

Vous pouvez vérifier que votre fichier CSS n'a pas d'erreur de syntaxe en cliquant sur l'icône , en sélectionnant dans la barre de menu **XML** puis **Check this CSS** ou encore en utilisant le raccourci clavier Ctrl + K.



La possibilité de vérifier qu'un fichier CSS ne comporte aucune erreur de syntaxe n'est pas offerte dans la version gratuite d'Editix. Si vous souhaitez utiliser cette fonctionnalité, il vous faudra passer sur la version payante.

Vous devriez normalement avoir un message indiquant que votre CSS est correct, comme l'illustre la figure suivante.




Lorsque tout est correct, ouvrez votre fichier XML dans un navigateur web pour observer le résultat.

## TP : mise en forme d'un répertoire téléphonique

Le but de ce TP est de créer un fichier CSS afin de mettre en forme un répertoire téléphonique se présentant sous la forme d'un document XML.

### Le document XML

Voici le document XML à mettre en forme :

**Code : XML**

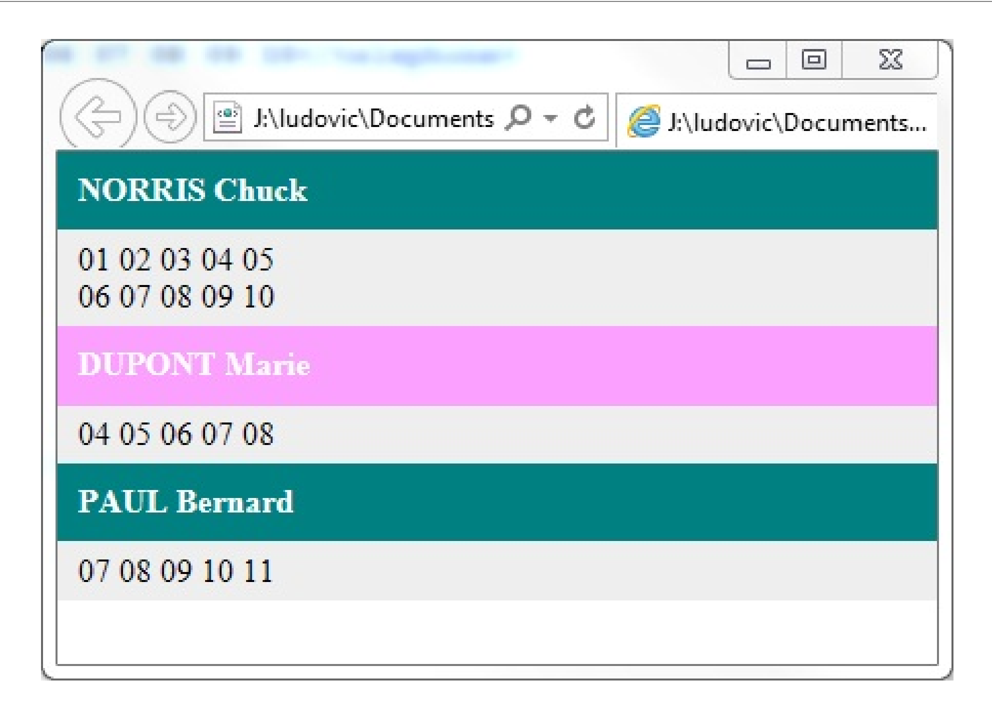
```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<personnes>
    <personne sexe="masculin">
        <identite>
     <nom>NORRIS</nom>
     <prenom>Chuck</prenom>
 </identite>
 <telephones>
            <telephone type="fixe">01 02 03 04 05</telephone>
            <telephone type="portable">06 07 08 09 10</telephone>
 </telephones>
    </personne>

    <personne sexe="feminin">
 <identite>
     <nom>DUPONT</nom>
     <prenom>Marie</prenom>
 </identite>
 <telephones>
            <telephone type="bureau">04 05 06 07 08</telephone>
 </telephones>
    </personne>

    <personne sexe="masculin">
        <identite>
     <nom>PAUL</nom>
     <prenom>Bernard</prenom>
 </identite>
 <telephones>
            <telephone type="portable">07 08 09 10 11</telephone>
 </telephones>
    </personne>
</personnes>
```

### La mise en forme

Voici en figure suivante une capture d'écran de la mise en forme que vous devez reproduire.




Comme vous pouvez le constater, l'identité d'une personne de sexe masculin est sur fond bleu tandis que celle d'une personne de sexe féminin est sur fond rose. Tous les numéros de téléphone sont sur fond gris.

## Une solution

Pour ne pas changer, voici un exemple de solution.

Le fichier XML avec le fichier de style CSS référencé :

**Code : XML**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="personnes.css" type="text/css" ?>
<personnes>
    <personne sexe="masculin">
 <identite>
     <nom>NORRIS</nom>
     <prenom>Chuck</prenom>
 </identite>
 <telephones>
           <telephone type="fixe">01 02 03 04 05</telephone>
           <telephone type="portable">06 07 08 09 10</telephone>
 </telephones>
    </personne>

    <personne sexe="feminin">
 <identite>
     <nom>DUPONT</nom>
     <prenom>Marie</prenom>
 </identite>
 <telephones>
           <telephone type="bureau">04 05 06 07 08</telephone>
 </telephones>
    </personne>

    <personne sexe="masculin">
        <identite>
     <nom>PAUL</nom>
     <prenom>Bernard</prenom>
 </identite>
 <telephones>
           <telephone type="portable">07 08 09 10 11</telephone>
 </telephones>
```

```
    </personne>
</personnes>
```

Le fichier CSS :

**Code : CSS**

```
personne {
    display:block;
}

identite {
    display:block;
    padding:10px;
    font-weight:bold;
    color:white;
}

personne[sexe="masculin"] identite {
    background-color:#008080;
}

personne[sexe="feminin"] identite {
    background-color:#FBA0FE;
}

telephones {
    display:block;
    background-color:#EEE;
    padding: 5px 10px;
}

telephone {
    display:block;
}
```

Ce tutoriel est en cours d'écriture, n'hésitez pas à passer régulièrement pour voir les nouveautés ! 

Un grand merci à Fumble pour ses relectures, ses conseils et ses encouragements tout au long de la rédaction de ce cours.

Je souhaite également remercier AnnaStretter pour ses conseils afin de rendre ce cours plus pédagogique.